Storyboard Programming of Data Structure Manipulations

A picture is worth 20 lines of code

by

Rishabh Singh

B.Tech(H), Indian Institute of Technology Kharagpur (2008)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

Author .....
Department of Electrical Engineering and Computer Science
May 7, 2010

Certified by .....
Armando Solar-Lezama
Assistant Professor
Thesis Supervisor

Accepted by .....
Terry Orlando
Chairman, Department Committee on Graduate Students

Storyboard Programming of Data Structure Manipulations
A picture is worth 20 lines of code

by

Rishabh Singh

Submitted to the Department of Electrical Engineering and Computer Science
on May 7, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

We introduce *Storyboard Programming*, a new programming model that harnesses the programmer's visual intuition about a problem to synthesize a correct implementation. The motivation for our technique comes from the domain of data-structure manipulations. In this domain, programmers often think in terms of abstract graphical visualizations but have a hard time translating that intuition into low-level pointer manipulating code. We aim to bridge this gap and show that it is possible to derive the low-level implementation automatically from the graphical specifications with little additional input from the programmer.

The storyboard in our programming model consists of a series of scenarios which show how the data-structure evolves under different conditions. We present two novel algorithms to synthesize the code from the storyboards. The algorithms derive an abstract domain and a set of correctness conditions automatically from the storyboards. The synthesizer uses the abstract domain to perform abstraction guided combinatorial synthesis. The resulting program is guaranteed to satisfy the correctness conditions derived from the storyboard, and to conform to the high-level structure specified by the programmer.

We have implemented our framework successfully on top of the SKETCH system. Our implementation is capable of synthesizing several interesting data-structure manipulations such as insertion, deletion, rotation, reversal over linked list and binary search tree data structures.

Thesis Supervisor: Armando Solar-Lezama
Title: Assistant Professor

Acknowledgments

I would first like to thank my advisor Prof. Armando Solar-Lezama to have put in so much efforts in making this thesis look what it is. He has always been an inspiration and a great motivator for helping me think about the big picture of our research. Also for the endless times I went to his offices for various issues with Sketch encoding and my understanding about it.

Prof. Daniel Jackson influenced a lot about working on big important ideas. My undergraduate advisor Prof. Andrey Rybalchenko who made me believe that everything in life is acheivable, its just that we need to work that bit harder. In his words: *Believe to Achieve.* I have been trying to do that ever since.

I was fortunate enough to work during internship with great researchers like Dimitra Giannakopoulou, Corina Pasareanu and Tom Henzinger. They helped me ever so much at every point with all different kinds of issues.

My fellow CSAIL graduate students Eunsuk Kang, Sasa Misailovic, Aleksandar Milicevic, Joseph P. Near, Jean Yang, and Kuat Yessenov have never failed to provide inspiring discussion and helpful feedback. Also my roommates Saurav Bandopadhyay and Rahul Rithe never let me miss my home too much.

I would like to thank my father, its all because of him and his wishes I am where I am today. I would also like to thank my mother, my sister Richa and my brother Rohit for all the sufferings they took for me. Finally I would like to specially thank Deeti for helping me ever so much at every point of my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Storyboard programming is a new programming paradigm for programmers to automatically synthesize efficient low level code from intuitive high level graphical specifications. The motivation of the storyboard programming model comes from the domain of data structure manipulations. In this domain, programmers have a very clear picture of how a data structure evolves and gets modified during the execution. This visual intuition is then required to be implemented for executing the desired manipulation. The correct implementation of such abstract graphical intuitions in low level programming languages requires a lot of precise reasoning and is often troublesome. Storyboard programming aims to reduce the burden on programmers by bridging this gap between the high level intuitive graphical specifications and the corresponding low level pointer manipulating implementation.

For providing unbounded correctness guarantees of the synthesized implementation over the size of the data structures, the synthesizer requires an abstract domain. The abstract domain over-approximates an infinite set of concrete configurations of a data structure to a finite set of abstract configurations. An important technique storyboard programming enables is the automated derivation of the abstract domain from the graphical specifications.

1.1 Bridging the gap between graphical intuition and implementation

Consider the example of binary search tree rotation in figure 1-1. As can be seen from the figure, it is quite natural and intuitive to express the rotation manipulation on binary search trees graphically in terms of a starting configuration and an ending configuration of the tree. In fact this is a standard way a computer science undergraduate student learns about data structure manipulations in an Algorithms class. On the other hand consider the corresponding low level code for the rotation example [6] shown in figure 1-2. Even though figures 1-1 and 1-2 essentially capture the same notion of tree rotation operation, the correspondence between the two is not straightforward. The code for implementation requires a precise sequence of pointer assignments to carry out the required rotation operation. These low level implemen-

tations are non intuituive and in some cases can get very complicated; e.g. in the case of red black tree manipulations.

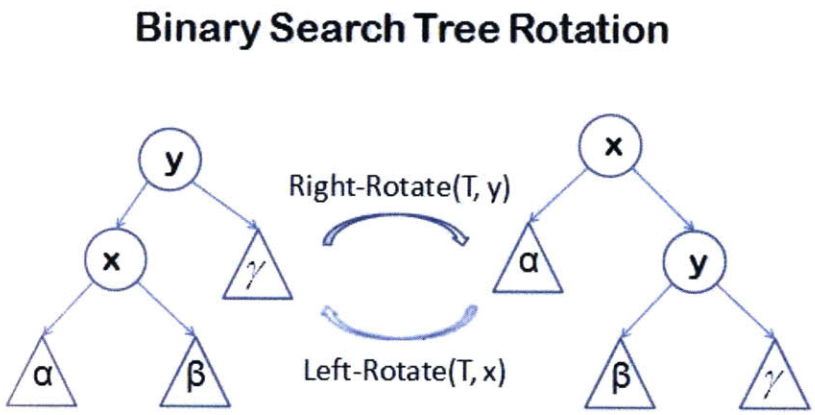

Figure 1-1: Binary Search Tree rotation

1.2 The promise of graphics aided programming

The use of graphics to aid in programming, debugging and program understanding has historically been an intriguing prospect. Myers [17] presents a broad survey of various attempts at using high-level graphical descriptions to alleviate some of the complexity of programming. Most of those techniques can be broadly classified into three main classes : Visual Programming, Programming by example and batch/interactive programming. AMBIT/G [5] was one of the earliest efforts for representing data and programs as predefined pictures and using a pattern matching language to execute them. Grail [8] could compile flow chart programs directly to executable code. Some systems like Shaw's [21] were proposed that could learn some restricted set of programs from input/output pairs. Programming visualization systems [16] [4] were also developed to display runtime data structure information for providing help in debugging of the programs. Several other systems like Pygmalion [22] and Thinglab [3] were developed to help programmer define computations pictorially. Programming Languages like Visual Basic allowed programmers to create GUI applications from general graphical components using drag-and-drop techniques. Most of these systems either require programmers to know about the how the implementation works (visual programming) or only try to infer restricted set of arbitrary programs (programming by example). Despite such early enthusiasm, the graphics aided programming has had a very limited impact for most programming purposes. Most of the previous attempts did not attempt to exploit the semantic information present in the pictures, rather used them only syntactically as a means of communication between the programmer and the machine. With the recent advancement of verification and SAT technology,

Algorithm 1 LeftRotate(Node root, Node x)

```
 1: y = x.right
 2: x.right = y.left
 3: if y.left ≠ null then
 4:    y.left.p = x
 5: end if
 6: y.p = x.p
 7: if  x.p == null then
 8:    root = y
 9: else
10:    if x == x.p.left then
11:       x.p.left = y
12:    else
13:       x.p.right = y
14:    end if
15: end if
16: y.left = x
17: x.p = y
```

Figure 1-2: Binary Search Tree rotation code

we aim to push the graphical aided programming usage to the next level so that programmers can benefit from the large amount of computing resources available today.

1.3 The Storyboard Programming Framework

Storyboard programming facilitates programmers to represent various cases of the data structure modifications graphically using the storyboards. Each case consists of abstract input and output (possibly intermediate) data structure configurations that ideally represent a distinct behaviour of the manipulation. The synthesized low level implementation conforms to these graphical input/output specifications. These visual specifications are then mapped to constraints in the storyboard language. In addition to the storyboard, the programmer also provides some approximate control flow of the required implementation. This control flow sketch is required to structure the search space for the correct implementation and to inhibit the synthesizer from synthesizing non-structured implementations. The framework derives an abstract domain and a set of correctness conditions from the storyboards, which in conjunction with the control flow sketch are reduced to a set of equations in the intermediate SKETCH language. The synthesized solution to the equations is then mapped back to the desired low level implemenation corresponding to the storyboards.

1.4 From verification to synthesis

The framework synthesizes the desired implementation in two steps. In the first step, an abstract domain and a set of correctness constraints are automatically derived from the storyboard. The abstract domain is required for providing unbounded correctness guarantees for the resulting implementation in terms of size of the data structure. The correctness constraints correspond to the input, intermediate and output state constraints obtained from the graphical specifications.

In the second step, a set of data flow equations of the form $X = f(X)$ are dervied from the control flow sketch provided by the programmer. This approach of translating the program into a system of $O(n)$ equations where n is the number of program points is very similar to the approach that maps a program into a monotone framework [18]. For the synthesis problem, the program statements are unknown but the set of input and output states are known. The task of the synthesizer is to efficiently search for a set of program statements that conform to the data flow, the input/output constraints and possibly intermediate state constraints. In this research, we explore two novel ways to search for functions in the equations from its solution. The first approach EXPSYN constructs the states and transition functions explicitly whereas the second approach IMPSYN models them symbolically.

We evaluate the performance of the two algorithms on case studies involving manipulations like insertion, deletion, rotation and reversal for the linked list and binary search tree data structures. We first wish to explore how far this framework can be used to synthesize the well known data structure algorithms. The future goal of this research is to use this framework for synthesizing implementations for manipulations involving arbitrary user defined data structures.

1.5 Contributions

This thesis makes the following key contributions:

- Development of storyboard programming as a new programming paradigm.
- Automated abstract domain derivation from the storyboards to provide unbounded correctness guarantees.
- Two novel synthesis algorithms EXPSYN and IMPSYN to solve the data flow equations.
- Storyboard language to formally specify the meaning of storyboards.
- Evaluation of the algorithms on a handful of data structure manipulations including linked lists and binary search trees.

1.6 Organization of the thesis

In chapter 2, we describe the overview of the storyboard programming framework through two running examples. Chapter 3 describes the preliminaries about the SKETCH synthesis system. Chapter 4 presents the syntax and semantics of our storyboard language. Chapter 5 and 6 present the detailed description of EXPSYN and IMPSYN algorithms respectively. Chapter 7 presents the evaluation of the two algorithms on case studies including various manipulations on linked list and binary search tree data structures. Chapter 8 describes some of the relationship between our work and the work on visual programming, programming by example and some recent software synthesis work. Finally Chapter 9 presents the conclusions and the future work directions.

Chapter 2

Storyboard Programming Overview

This chapter presents a brief overview of the Storyboard programming framework. We present two interesting classes of data structure manipulations through two representative examples: insertion in a sorted linked list and inplace linked list reversal. We briefly describe how the programmer describes the storyboards and how the framework derives an abstract domain and correctness conditions for synthesizing the required implementation.

2.1 Scan and Modify manipulations: linked list insertion

We present the first flavour of storyboard programming with a simple sorted linked list insertion example. This example is representative of a class of data structure manipulations which first require a scan over the data structure and then a local modification to acheive the desired goal. The insertion operation requires to insert a node x in a sorted linked list l that may possibly be empty.

2.1.1 Storyboard

As a first step, the programmer provides his visual insights using the storyboard as shown in fig 2-1. The storyboard consists of a set of abstract input/output pairs, possibly with some intermediate states. The storyboard in figure 2-1 consists of four different *scenarios*; where a scenario describes how a data structure evolves and gets modified under a certain condition. The four scenarios for the example respectively describe: inserting a node in the middle of the list, inserting a node at the beginning of the list, inserting a node at the end of the list and inserting into an empty list.

The insertion manipulation requires a node x to be inserted in the list between two nodes a and b (possibly empty) with the invariant a.val $<$ x.val $<$ b.val. Therefore only the concrete nodes a and b are relevant for inserting x at the correct position and other nodes are abstracted away. The programmer visualizes the sorted linked list in four parts: front, a, b and back. The node front represents a set of nodes which are in front of the node a and back represents a set of nodes which are present after

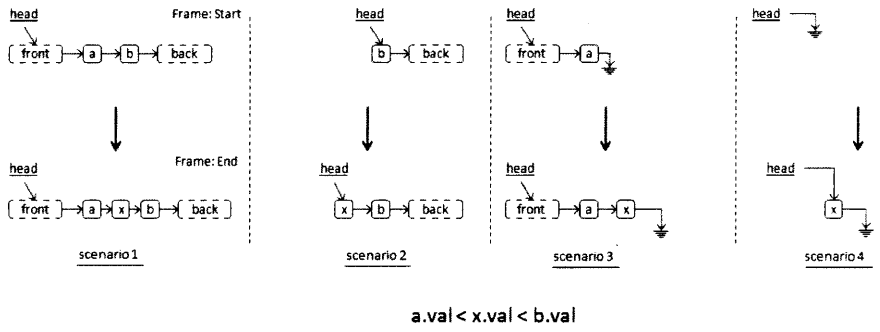

Figure 2-1: Storyboard for linked list insertion

the node b. We call such nodes *abstract* nodes that represent a set of concrete nodes. Nodes a, x and b on the other hand are concrete nodes.

2.1.2 Control flow sketch

In addition to the storyboards, the programmer provides a control flow sketch to structure the search space for the desired implementation. The control flow sketch for the linked list insertion example is shown in figure 2-2. As a first step, the programmer describes the struct Node of the linked list. In the control flow sketch, LOC describes all possible expressions over nodes that can be obtained with at most one dereferencing with the next pointer. The programmer also defines possible choices for conditionals COND that might be useful for the program. In many cases, these conditionals can be derived automatically from different scenarios as they essentially represent different cases for the program execution. An assignment statement is described as STMT: LOC = LOC and a conditional statement as CSTMT: if(COND) STMT. It can be noted that STMT defines all possible assignment statements over variable expressions with at most one next dereferencing operation. (STMT)* denotes zero or more number of STMT statements.

The control flow sketch in figure 2-2 describes first a sequence of assignment statements (STMT)*, then a while loop with the loop body consisting of a set of assignment statements and finally a set of conditional statements. A large part of the control flow sketch that is provided by the programmer can be easily automated, but our current framework requires sketches to be provided manually. Ideally the framework should only require some measure of the algorithmic complexity of the required implementation for automated construction of the control flow sketch. The control flow sketch is then translated into a system of equations which are then solved in an abstract setting.

In order to provide unbounded correctness guarantees over the synthesized implementation, an abstract domain is derived from the storyboards. The abstraction reduces the infinite set of inputs to a small finite number of abstract inputs that

```
Node{
  int value;
  Node next;
  invariant next.value >= value;
}
```

```
void insert(Node head, Node x){
/*@Start*/
   Node prev, curr;
   (STMT)*
   while(COND)
      (STMT)*
   (CSTMT)*
/*@End*/
}
```

Figure 2-2: Control flow sketch for linked list insertion

makes the synthesis algorithm efficient and feasible.

2.1.3 Derivation of the abstract domain

The synthesizer derives an abstract domain automatically from the storyboard provided by the programmer. For each scenario, the system defines an abstract domain focusing on the concrete and abstract objects in that scenario. To illustrate the dervivation, we consider the first scenario of inserting a node in the middle of the list in figure 2-1. The graphical specification in the figure can be translated into a concise text based notation in the storyboard language. All the algorithms we define later will be defined in terms of this notation, but we want to emphasize that this translation from graphical representations to the text based language is straightforward. The textual description consists of two parts: *environment* and *frames*. The environment describes the set of objects involved in the scenario, whereas the frames define the description of data structure at particular program points of the corresponding implementation.

For the first scenario in figure 2-1, the environment is described as follows:

```
Env{
  Node head, prev, curr;
  [Node] a, b, x;
  [[Node]] front, back;
  assert front.next == {front, a};
  assert back.next == {back, null};
}
```

The environment first describes a set of variables of type Node that the programmer thinks might be useful. The local variables from the control flow sketch are included in this set of variables. The environment above describes three local variables head (from the storyboard), prev and curr (from control flow sketch). [Class] name* is used to define concrete objects of type Class whereas [[Class]] name* is used to define abstract objects of type Class. An abstract object refers to a collection of concrete objects sharing some common property. The environment for the example

above describes three concrete nodes a, b and x and two abstract nodes front and back. The environment also defines the global connectedness properties of the abstract nodes; specifically that front.next points to both front and a, and back.next points to back and null. These definitions are part of the environment as they hold for all scenarios and do not get modified for this class of examples.

After defining the environment, the next step is to define the frames. Frames represent the description of the data structure at different points in the execution. For the first scenario, we have two frames. One frame corresponding to the beginning of the execution and the other frame corresponding to the end of the execution. A programmer could add multiple other frames possibly describing the invariants about the shape of data structure inside the loops. The Start and End frames describe the various pointer assignments in the corresponding graphical representations in a textual form as described below.

```
Start{
    head = front; a.next = b; b.next = back;
}

End{
    head = front; a.next = x; x.next = b; b.next = back;
}
```

We need to compile the description of storyboards in the above language to an abstract domain. We use predicate abstraction (powerset) [1] as our abstract domain. The predicates for our abstract domain are algorithmically derived from the storyboards itself. A predicate corresponds to an assignment of a variable to some concrete location in the storyboard. The synthesizer considers all updatable pointers and variables. In this example, we have head,prev,curr (local variables) and a.next, x.next and b.next as potential updatable variables. There are six locations in the storyboard that they can point to: front, a, x, b, back and null.

An abstract state is defined as a set of predicates, i.e. it represents a valuation of the variables to the locations. An example abstract state for the above scenario would be s = { head = front, prev = front, curr = b, a.next = b, b.next = back, x.next = null }. Since we have 6 updatable variables and 6 different locations, we get 6^6 possible abstract states which is quite large. The lattice for our abstract domain consists of sets of such abstract states with the order defined in terms of the subset relationship. We will name such set of abstract states as *abstract set-states* for disambiguating them with an abstract state. An abstract set-state in this domain can be conviniently represented as a bitvector of states, where a set bit implies the reachability of the corresponding state at a program location. The synthesis problem is finally reduced to the problem of satisfying a set of constraint equations. The bitvector representation helps us in defining the abstract interpretation [7] over these equations as a SAT problem.

2.1.4 Translation to constraint equations

We first briefly describe the main ideas behind constraint based abstract interpretation [9]. Then we use a similar approach to use the abstract domain derived from the storyboards to generate constraints for the possible implementations, which are then used by the SAT solver to synthesize the desired implementation.

In order to perform abstract interpretation, we require the control flow graph (CFG) of the program. We obtain the CFG of possible implementations from the control flow sketch provided by the programmer. Figure 2-3 shows the control flow graph for the example. In the graph, the blocks f_1, f_2 and f_3 correspond to the missing blocks of code in the sketch provided by the programmer. The abstract set-states of the program at program points t_{in}, t_1, t_2, t_3 and t_{out} are defined by the following equations :

$$t_1 = f_1(t_{in}) | f_2(t_2) \tag{2.1}$$
$$t_2 = f_c(t_1) \tag{2.2}$$
$$t_3 = f_{\bar{c}}(t_1) \tag{2.3}$$
$$t_{out} = f_3(t_3) \tag{2.4}$$

where every function f_i takes as input a set of program states and maps it back to the corresponding set of output program states.

Let us consider for a moment that the sketch is a complete program without any unknown statements. In this case, the functions f_1, f_2 and f_3 are the transitions functions for each block of the code. After substituting t_{in} and the functions f_i in the above equation, the least fixpoint solution of the equation gives us the most accurate abstraction of the reachable set of concrete states at every program point [12].

For our synthesis problem, both the set of input and the corresponding set of output states t_{in} and t_{out} are known, but the functions f_i are all unknown. The synthesizer searches for an assignment of these functions which satisfies the above set of equations after substituting the appropriate values for t_{in} and t_{out}. From the sketch of the implementation in section 2.1.2, we can see that possible choices for f_1 and f_2 are over a finite number of STMT functions. We assume that the function blocks have a bounded length which can be algorithmically varied to discover the required bounds. The possible choices for f_3 are over a set of finite conditional statements CSTMT and the possible choices for f_c are over a finite set of conditions COND. We encode these finite choices for every function f_i with an additional control parameter $c_{??}$ in a similar spirit to the SKETCH system [23]. The value of the control parameters represents the choice made by the synthesizer for its corresponding function. The data flow equations now become :

$$t_1 = f_1(t_{in}, c_{??}) | f_2(t_2, c_{??}) \tag{2.5}$$
$$t_2 = f_c(t_1, c_{??}) \tag{2.6}$$

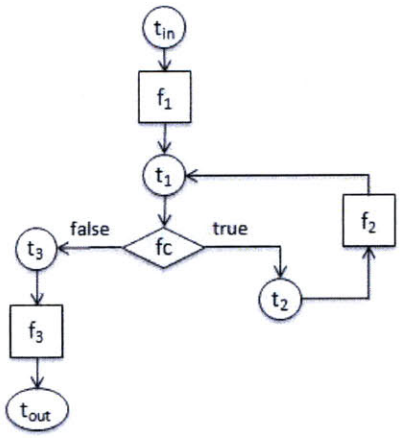

Figure 2-3: Control flow graph for linked list insertion

$$t_3 = f_{\bar{c}}(t_1, c_{??}) \tag{2.7}$$
$$t_{out} = f_3(t_3, c_{??}) \tag{2.8}$$

We finally need to encode the correctness condition which comes from the storyboard scenarios. We impose the constraint that t_{in} and t_{out} are the set of input (S_{in}) and output states (S_{out}) respectively as presented in the storyboard.

$$t_{in} = S_{in} \tag{2.9}$$
$$t_{out} = S_{out} \tag{2.10}$$

Any solution to the set of equations obtained from the data flow constraints and the input/output constraints is a potential candidate for the required implemenation. We note that we do not necessarily require the least fixed point solution to the above set of equations, as we are not necessarily interested in the valuation of the intermediate set-states t_i; we are interested in the values of the control parameters $c_{??}$. The value for control parameters is sufficient to fill up the missing blocks of code with the corresponding statement choice. By giving the above constraints to a SAT solver, we get a solution to our synthesis problem. The generated implementation is verified with respect to the specifications provided in the storyboard.

This class of data structure manipulations do not require abstract node updates; i.e. they are only used in the scanning phase but not in the modification phase. On the other hand some class of data structure manipulations do require abstract node updates for performing the desired operation. We describe about those class of manipulations with a representative example in the following section.

2.2 Handling abstract node updates : linked list reversal

Some data structure manipulations require modifying the abstract node pointers in the storyboard input/output specification. The inplace linked list reversal is a representative example of this class of data structure manipulations. For such cases, we need additional machinery in our framework to handle such abstract node updates. The machinery is described briefly in this section.

2.2.1 Storyboard

As in the previous example, the programmer provides an intuitive graphical specification of the linked list reversal operation. The storyboard in Figure 2-4 consists of four scenarios respectively: reversing a list with more than two elements, reversing a list with two elements, reversing a list with one element and reversing an empty list. Scenario 1 represents an abstract list with two concrete nodes a and b, and one abstract node mid.

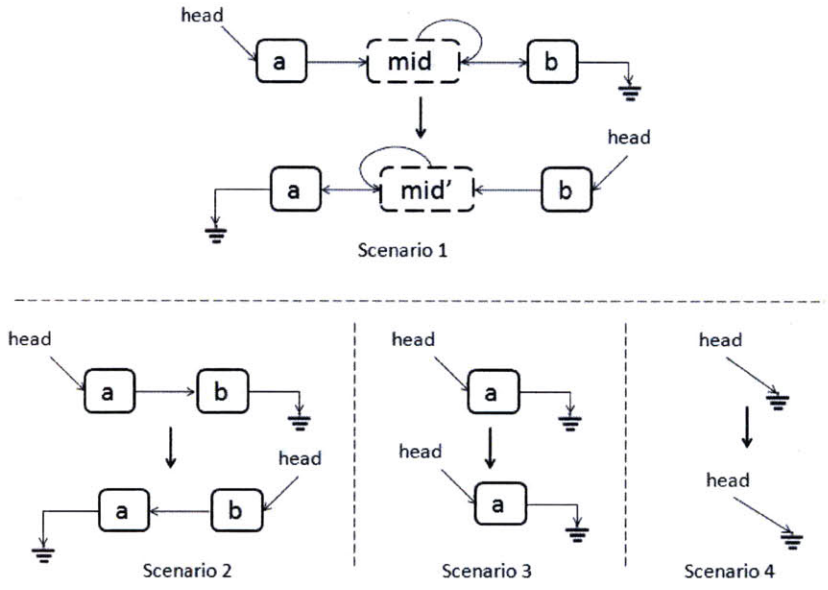

Figure 2-4: A Scenario in the storyboard for linked list reverse manipulation

In addition to the storyboard, the programmer provides a *fold* and an *unfold* operation on the abstract node mid as the reversal operation requires modification of the next pointer of the abstract node mid. The fold and unfold operations are shown below in figure 2-5. The unfold operation nondeterministically either expands the abstract node mid into a concrete node x pointing to abstract node mid or to a

concrete node x. The inverse operation fold collapses the node pattern of the concrete node x pointing to abstract node mid or a concrete node x to the abstract node mid.

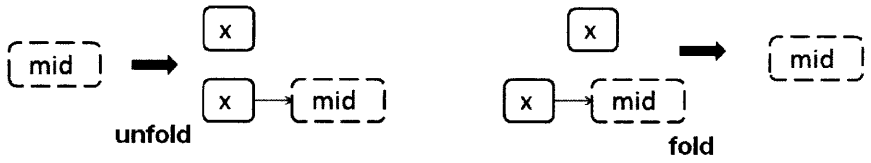

Figure 2-5: Unfold and fold operation on the mid abstract node

2.2.2 Control flow sketch

The control flow sketch for the in-place linked list reversal example is shown in figure 2-6. First, the programmer describes the struct Node of linked list. In the control flow sketch, LOC describes all possible nodes that can be obtained with at most one dereferencing with the next pointer. An assignment statement is described as STMT: LOC = LOC. It can be noted that STMT defines all possible assignment statements with at most one next dereferencing operation. (STMT)* denotes zero or more number of STMT statements. COND represents the choices of conditions which are inferred from different scenarios.

The control flow sketch in figure 2-6 describes a sequence of assignment statements (STMT)* followed by a while loop with the loop body consisting of a set of assignment statements. At the beginning of the loop body, we have an unfold method call which unfolds its argument node if it currently points to the mid node. Similarily the loop ends with a fold statement which folds its argument node if it currently points to the concrete node x. For this example, the framework only requires that the required algorithm has a linear ($O(n)$) complexity.

2.2.3 Derivation of the abstract domain

The graphical specification from the scenario is translated in the text based notation of the storyboard language, which as described before consists of an *environment* and *frames*. The environment for this example is described as follows:

```
Env{
  Node head, temp1, temp2, temp3;
  [Node] a, x, b;
  [[Node]] mid, mid';
  fold mid x {true, x.next = mid};
  unfold x mid' {true}

}
```

```
Node{
  int value;
  Node next;
}
```

```
void insert(Node head, Node x){
/*@Start*/
   Node temp1, temp2;
   (STMT)*
   while(COND){
      fold(LOC);
      (STMT)*
      unfold(LOC);
      }
/*@End*/
}
```

Figure 2-6: Control flow sketch for in-place linked list reversal

Since the abstract node mid is getting modified during the reversal operation, the abstract node gets unfolded and folded several times during the execution. The programmer provides some bounds on the number of time it is expected to be unfolded before reaching a convergence. In the example above, the concrete node x denotes the node showing up due to the unfold operation. The environment describes four updatable variables head,$temp_1$, $temp_2$, $temp_3$ and seven pointer locations a.next, x.next, $mid.next_1$, $mid.next_2$, $mid'.next_1$, $mid'.next_2$ and b.next. The abstract nodes mid and mid' have two next pointers $next_1$ and $next_2$ as specified in the graphical specification. The multiple next pointers for abstract nodes correspond to the non-deterministic choice made by the next dereference. For this example, we have 11 updatable variables and 4 concrete locations which gives us 11^4 abstract states.

It can be noted that the next pointer definitions of the abstract nodes are missing from the environment. This is the case because they do not hold the same values throughout the execution unlike the previous class of manipulations. The environment also defines the fold and unfold operations. The syntax of the unfold statement defines that if its argument node matches node mid, then node mid should be replaced with node x and nondeterministically either no new constraints are enforced (true) or the constraint x.next = mid is added. Similarily the fold function declaration defines that if its argument node matches node x, then x should be replaced with node mid' with no added constraints.

2.2.4 Translation to constraint equations

The algorithm first builds the control flow graph (CFG) from the control flow sketch as previously. The CFG for the example is shown in figure 2-7. The functions f_{fold} and f_{unfold} take an additional argument node (LOC) as an input parameter to match it with the node specified in the environment definition. The data flow equations are then derived from the CFG as shown below:

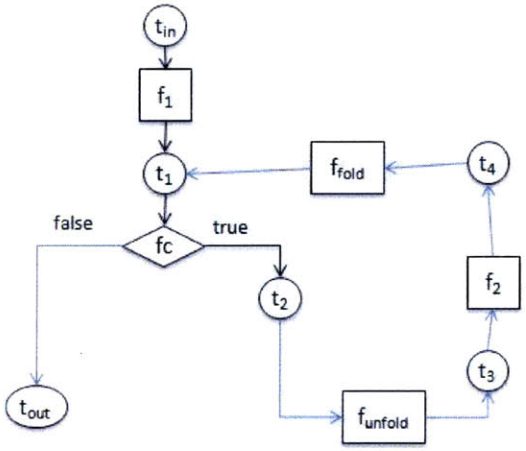

Figure 2-7: Control flow graph for in-place linked list reversal

$$t_1 = f_1(t_{in}) | f_{fold}(LOC, t_4) \tag{2.11}$$
$$t_2 = f_c(t_1) \tag{2.12}$$
$$t_3 = f_{unfold}(LOC, t_2) \tag{2.13}$$
$$t_4 = f_2(t_3) \tag{2.14}$$
$$t_{out} = f_{\bar{c}}(t_1) \tag{2.15}$$

Each of the unknown functions f_i, f_{fold} and f_{unfold} can take one of the finite possible choices as described by the sketch. We can augment the functions with the control choice parameter $c_{??}$ as previously to obtain the following data flow equations:

$$t_1 = f_1(t_{in}, c_{??}) | f_{fold}(LOC, t_4, c_{??}) \tag{2.16}$$
$$t_2 = f_c(t_1, c_{??}) \tag{2.17}$$
$$t_3 = f_{unfold}(LOC, t_2, c_{??}) \tag{2.18}$$
$$t_4 = f_2(t_3, c_{??}) \tag{2.19}$$
$$t_{out} = f_{\bar{c}}(t_1, c_{??}) \tag{2.20}$$

The correctness conditions are also encoded as in the previous case:

$$t_{in} = S_{in} \tag{2.21}$$
$$t_{out} = S_{out} \tag{2.22}$$

2.3 Solving the data flow equations

In this section we briefly describe two novel ways for solving the data flow equations obtained from the storyboard to synthesize the desired implementation. The first algorithm EXPSYN enumerates the abstract states and transition functions explicitly whereas the second algorithm IMPSYN represents the abstract states and transition functions symbolically.

2.3.1 EXPSYN algorithm

The set of data flow equations are singly existentially quantified and the algorithm solves them by translating them into a SAT satisfaction constraint problem. The algorithm first enumerates all the abstract states ($\mathcal{S}$) and constructs the transition functions mapping an abstract set-state to another abstract set-state. These transition functions can be non deterministic due to the abstraction involved. An abstract set-state at any program location t_i is represented as a bitvector of size $|\mathcal{S}|$. The j^{th} set bit of the bitvector t_i represents the reachability of the state s_j at program location i. The transition functions map a bitvector to another bitvector. The input state and output state bitvectors t_{in} and t_{out} are derived from the storyboard. The resulting constraint is solved by an off the shelf SAT solver. It can be observed that this approach might not scale very well as this encoding is doubly exponential in the number of abstract states ($|\mathcal{S}|$). But from our experience, this simple algorithm is still useful for simple manipulations.

2.3.2 IMPSYN algorithm

The IMPSYN algorithm represents large state spaces symbolically instead of treating them explicitly. This helps us in avoiding computation of a large number of wasteful unreachable states, as it is often the case that the reachable set of abstract states is considerably smaller. This, though, puts the burden of symbolic computation of the transition functions towards the solver side but in turn provides us the ability to handle very large state spaces. The algorithm trades some of the time complexiy with memory complexity as the memory requirements now are much smaller in comparison.

The algorithm divides the task of solving the constraint equations into two phases: Verification phase and Synthesis phase as shown in Figure 2-8, in the same spirit as CEGIS [23]. Here instead of using counterexample inputs, the algorithm uses counterexample paths for inductive synthesis. The non-determinism in the execution comes from the presence of abstract nodes. The Verifier in the verification phase is an abstract interpreter that searches over the abstract state space to verify that only correct final states are reachable at the return location. If not, it returns a deterministic counterexample path to the Synthesis phase. But the synthesis phase only consists of these deterministic counterexample program paths as constraints and thereby only needs to maintain one abstract state at each program point in contrast to mainting set-states previously. These paths are also solved incrementally thereby reducing the memory and time overhead of combined solving. This loop between

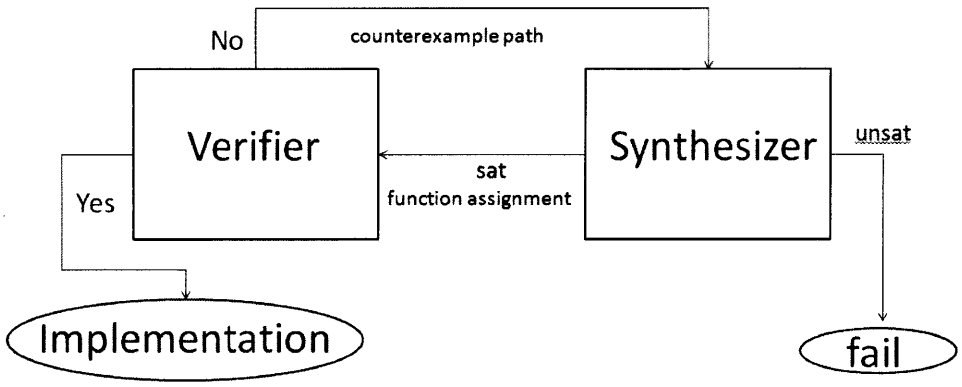

Figure 2-8: The IMPSYN algorithm

Synthesis and Verification phases continues until the Verifier proves the synthesized program works correctly.

Chapter 3

Preliminaries

This chapter presents a brief description about the SKETCH synthesis system which would be useful to understand the later chapters on EXPSYN and IMPSYN algorithms.

3.1 The SKETCH Synthesizer

The SKETCH synthesis framework was originally applied to bit-stream manipulations [26] and has recently been applied to scientific programs [24] and concurrent datastructures [25]. The SKETCH system is pushing the frontier towards making the automated program synthesis approach practical and useful for general programming purposes by harnessing programmer insights about the problems.

The input language for SKETCH is a subset of C language with built-in support for structs, integer and bitvector arrays, assert statements and some special sketching constructs which are briefly described below. It requires a sketch of the partial program and its specification either in the form of some inefficient implementation or in the form of some assert statements in the sketch itself. The sketch allows the programmer to provide valuable insights to guide the search for synthesizing the correct implementation. The SKETCH system employs a Counterexample guided inductive synthesis algorithm (CEGIS) which iteratively adds only the interesting inputs for the program to be synthesized and ignores the ones which do not introduce any new behaviours.

3.1.1 Example sketch

We use a very simple sketch of a toy program in figure 3-1 to describe various useful features of the SKETCH language. The implements keyword specifies the specification implementation for the sketch program. The specification describes a function $f(x) = x + x = 2 * x$. The sketch describes a function $g(x) = x*??$, where ?? (integer hole) is a special keyword in the SKETCH language that can be filled with an integer value (upto a bounded size). For this example the integer hole will get filled up with value 2.

```
int spec(int x){
  return x + x;
}
```

```
int sketch(int x) implements spec{
  return x * ??;
}
```

Figure 3-1: A simple sketch toy program

Another important construct the SKETCH language provides is the choice construct which tells the synthesizer about the possible choices for a given location in the sketch program. The choice construct is specified with {| and |} symbols. Any regular expression over the possible choices can be written inside the choice constructs. Consider the problem of swapping two integer values x and y without using a temporary variable. The sketch in figure 3-2 provides the sketch using the choice construct.

```
int[2] swap(int x, int y){
  int[2] out;

  int temp;
  temp = x;
  x = y;
  y = temp;

  out[0] = x;
  out[1] = y;

  return out;

}
```

```
#define VAR {| x | y |}

int[2] sketch(int x, int y) implements swap{
  int[2] out;

  repeat(??){
     VAR = {| VAR + VAR | VAR - VAR |};
  }

  out[0] = x;
  out[0] = y;

  return out;
}
```

Figure 3-2: Swapping of two integers without a temporary variable

The macro VAR denotes the choice of variable which can either take values x or y. The construct repeat(n) is a special construct in SKETCH which lets the synthesizer repeat the block of statements inside its body n number of times. In the sketch the argument is specified as an integer hole and the synthesizer comes up with the minimum integer value sufficient for synthesizing the sketch. The body of the repeat construct contains another choice construct for the statement which can either be VAR = VAR + VAR or VAR = VAR - VAR. The synthesizer comes up with the following sequence of operations :

$$x = x - y; \tag{3.1}$$

$$y = x + y; \tag{3.2}$$

$$x = y - x; \tag{3.3}$$

which can be verified to correctly swap the two integer variables `x` and `y`. A detailed tutorial and comprehensive set of examples for using the SKETCH system can be found at SKETCH homepage.

Chapter 4

Storyboard Language

Storyboards need to have well defined semantics for them to serve as an effective means of communication between the programmer and the synthesizer. In this chapter, we provide the detailed description of the syntax of storyboard language and present the semantics of the language for efficiently deriving the specifications and abstract domains from the storyboard.

4.1 Syntax

$$
\begin{array}{rcl}
\textit{SBoard} & = & \textit{Scenario}^{+} \\
\textit{Scenario} & = & \textit{SLabel Env Frame}^{+} \\
\textit{Frame} & = & \textit{FLabel Constraint} \\
\textit{Constraint} & = & \textit{Pred;}^{*} \\
\textit{Pred} & = & \textit{VarName} = \textbf{loc} \\
 & | & \textbf{true} \\
 & | & \textbf{false} \\
\textit{Env} & = & \textit{Decl}^{*}\ \textit{Def;}^{*}\ \textit{FuncDecl}^{*} \\
\textit{Decl} & = & \textit{Type}\ \textbf{loc}^{+} \\
\textit{Def} & = & \textbf{assert}\ \textit{VarName} == \textbf{loc} \\
 & | & \textbf{assert}\ \textit{VarName} == \{\textbf{loc}_1,\textbf{loc}_2,\cdots,\textbf{loc}_n\} \\
\textit{FuncDecl} & = & \textit{FoldFuncDecl} \\
 & | & \textit{UnfoldFuncDecl} \\
\textit{FoldFuncDecl} & = & \textbf{fold}\ \textit{VarName VarName Constraint} \\
\textit{UnfoldFuncDecl} & = & \textbf{unfold}\ \textit{VarName VarName Constraint} \\
\textit{Type} & \in & \textit{Class} \mid \textit{[Class]} \mid \textit{[[Class]]} \\
\textit{VarName} & = & \textbf{var} \mid \textbf{object .field}
\end{array}
$$

Figure 4-1: Grammar for the Storyboard Language

Figure 4-1 presents the grammar of the storyboard language. We will use the

storyboard language description of the linked list insertion example in section 2.1.3 to explain the syntax of the storyboard language. A Storyboard *SBoard* consists of a set of one or more scenarios *Scenario*. In the example, we have 4 scenarios in the storyboard. Every scenario consists of a scenario label *SLabel*, an environment *Env* and a set of one or more frames *Frame*. Let us consider the scenario S_1 of inserting an element in the middle of a sorted linked list. The environmnet for S_1 is described as :

```
Env{
  Node head, prev, curr;
  [Node] a, b, x;
  [[Node]] front, back;
  assert front.next == {front, a};
  assert back.next == {back, null};
}
```

An evnironment *Env* consists of a series of variable declarations *Decl*, a set of global definitions *Def* and a set of function declarations *FuncDecl*. The declarations define class variables (*Class*), concrete objects (*[Class]*) and abstract objects (*[[Class]]*) of type Class. The evniroment of S_1 declares head, prev and curr as variables of Node class; a, b and x as concrete objects and front and back as abstract objects of type Node. The declaration follows a series of global definitions, initiated by the keyword assert, which hold for all scenarios. The environment for S_1 defines the next pointer locations for the abstract nodes front and back. The definition front.next == {front, a} defines that the next pointer of the front abstract node can point to either front itself or to the concrete object a. The function declarations define the fold and unfold operations over the abstract nodes. The function declaration starts with a keyword fold (or unfold) which is followed by two variable names var and var' and a set of constraints C. The syntax defines that when the argument to the fold (or unfold) method matches node var, it should be replaced with node var' and any one of the constraint in C can be asserted nondeterministically.

After the environment declaration, a scenario defines a series of one or more frames *Frame*. Each frame consists of a frame label *Flabel* which matches some label for a control location in the control flow sketch of the program. The frame also consists of a frame constraint *Constraint* which represents the constraint satisfied by the abstract state at the corresponding program control location in the scenario. The frames for the example scenario S_1 is described asq:

```
Start{
   head = front; a.next = b; b.next = back;
}

End{
   head = front; a.next = x; x.next = b; b.next = back;
}
```

The frame labels Start and End correspond to the starting and return program locations in the control flow sketch of the program. The frame constraint consists of a sequence of predicates of the form $Expr$ = **name**, the conjunction of which is expected to be true at the corresponding program location. In the example frame, the frame Start describes that the constraint formed by conjunction of head = front, a.next = b and b.next = back holds at the start location of the program in this scenario. It is worth noting that the language supports frames to be specified at any arbitrary control location of the program. This is useful for programmers to express additional insights about different invariants about the data strcutre manipulation which might help scale the synthesis process.

4.2 Semantics

This section describes how our framework derives the essential abstract domain automatically from the storyboards. We define a function $Scenario_i \rightarrow ABS_i$ that derives an abstract domain ABS_i from a scenario $Scenario_i$. The abstract domain for the storyboard $\mathsf{ABS_{SB}}$ is constructed from the abstract domain of the constituent scenarios $\mathsf{ABS_{SB}} = \{ABS_i \mid Scenario_i \rightarrow ABS_i\}$, with one abstract domain per scenario. The abstraction for each scenario is defined as a four-tuple $\mathsf{ABS_i} = \{\mathsf{Preds}, \{\mathcal{S}\}, L \rightarrow \{\mathcal{S}\}, Def\}$. In this definition, Preds is the set of predicates inferred from the Environment (Env) and the frame constraints. $\mathcal{S}$ is the set of abstract states; each abstract state corresponds to a valuation of the variables and object fields in the environment. Each state is associated with the set of predicates that are true in that state, $\mathcal{S} \subseteq \mathcal{P}(\mathsf{Preds})$. Finally, $L \rightarrow \{\mathcal{S}\}$ represents the mapping from the labels L to their corresponding set of states, and Def represents the definitions from the environment.

We need some additional definitions to describe the automated abstract domain derivation from the storyboards. Each scenario in the storyboard is modelled as a bipartite graph $G = (V, E)$, called the scenario constraint graph, where the two partitions of the vertices correspond to the set of updatable variables (V_V) and the set of concrete and abstract objects (V_L) respectively in that scenario. We have $V(G) = V_V \cup V_L$. For the example scenario S_1, we construct the following bipartite graph as shown in figure 4-2(a). The set of updatable variables V_V ={ head, prev, curr, a.next, b.next, x.next} and the set of abstract and concrete locations V_L = {front, a, x, b, back, null} constitutes the vertex set $V(G)$.

For every predicate of the form $u = v$ in a frame constraint in that scenario, we add an edge $e = (u, v) \in E(G)$ in the graph. For temporary variables from the control flow sketch, edges to all the locations are added in $E(G)$. This graph captures all possible variable valuations to the concrete and abstract locations for the corresponding scenario. In scenario S_1, all the edges are included in $E(G)$ for temporary variables prev and curr. It can be noted that for other variables like front, only one edge front $\rightarrow$ head is added as this is the only constraint present in the frames in S_1. Similarily for a.next only the edges to nodes b and x are present.

We define a *state edge cover* of this graph G to be a set of edges $E_{SEC} \subseteq E(G)$ such

that for every vertex $u \in V_V$, there exists an edge $(u, v) \in E_{SEC}$ for some $v \in V(G)$. In other words,

$$E_{SEC} = \{\ (u, v) \in E(G) \mid \forall u \in V_V\ \exists v \in V(G)\ \} \tag{4.1}$$

Furthmore restricting the size of the state edge cover set ($|E_{SEC}|$) to the number of updatable variables ($|V_V|$), we can guarantee that we have only one valuation of every variable in the edge cover set. i.e.

$$\forall u \in V_V\ \exists\ \textbf{one}\ v \in V(G)\ s.t.\ (u, v) \in E_{SEC} \tag{4.2}$$

A state edge cover set of size $|V_V|$ constitutes an abstract state for that scenario. The edges in the state edge cover represents the valuations of the variables in that particular set. Figure 4-2(b) presents an example state edge covering of G which in turn represents the abstract state s = { head = front, prev = front, curr = b, a.next = b, b.next = back, x.next = null }.

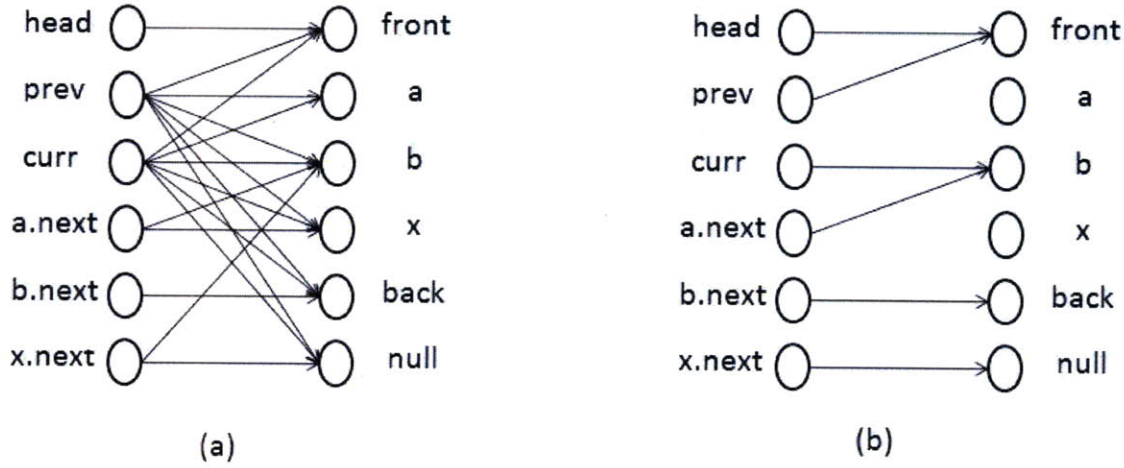

Figure 4-2: (a) Scenario constraint graph for the example scenario S_1 and (b) an example state edge cover (abstract state) for S_1

The state edge cover set computation in the scenario constraint graph performs an important optimization in which it merges states that are deemed unlikely to be relevant in synthesizing the correct implementation. The idea is that we need to only track the variable location valuations which are present in the storyboards. For the example scenario S_1, a.next points to b in the Start frame and points to x in the End frame. Our abstraction assumes that only b and x are relevant locations for a.next; so we only consider two predicates a.next = b and a.next = x instead of considering 6 different predicates a.next = {front,a,x,b,back,null}. It should be noted that for temporary variables (from control flow sketch) we still consider all 6 possible valuations. After performing similar optimizations for other updatable variables, we

get only 144 abstract states which are considerably smaller than 6^6. This optimization goes a long way in making the EXPSYN algorithm feasible. In a similar way, we derive an abstract domain for each scenario of the storyboard.

The formal semantics of the Storyboard language to obtain the abstract domains is presented in Figure 4-3. A high level description of the semantic rules follows:

- SB The rule for storyboards constructs an abstract domain for a storyboard by taking the union of abstract domains obtained from the abstract domains of the constituent scenarios.

- $SCEN$ The scenario rule constructs an abstract domain ABS for a scenario. From the environment of the scenario, we derive the variable bindings and the constituent definitions. From each frame $Frame_i$ with the variable binding, we construct a graph G_i using the frame rule ($FRAME$). The graph G for the scenario is then constructed by taking the union of graphs G_i.

- $FRAME$ The frame rule returns a Graph and the corresponding label from the constraints.

- $CONSTR$ The constraint rule constructs a graph $G = (V, E)$ from a set of constituent predicates. Every predicate contributes an edge and two vertices to the graph.

- $PRED$ The predicate rule converts a predicate of the form $(u = v)$ to two vertices u, v and an edge (u, v).

A Storyboard abstraction ABS_{SB} corresponds to a set of scenario abstractions (SB). For a scenario Sc, its correpsonding abstraction is defined as $ABS(Sc) = \{\mathsf{Preds}(G), S(G), Label \rightarrow S(G), Def\ \}$. For every frame $f_i \in Sc$, let G_i denote its constraint graph. The graph G for scenario is constructed as the union of all frame graphs, $G = \bigcup_i G_i$. $\mathsf{Preds}(G)$ is evaluated as the predicates corresponding to all the edges in G. $S(G)$ is the union of all possible variable edge covers of G. $Label \rightarrow S(G)$ is evaluated as the union $\bigcup_i L_i \rightarrow S(G_i)$ for every frame i. Finally the definitions Def are evaluated from the environment of the scenario ($SCEN$). A predicate $(u = v)$ is converted to its corresponding edge (u, v) and vertices u and v ($PRED$). The graph for a frame constraint is constructed from its constituent set of predicates ($CONSTR$). Using this graph and the frame label, the frame graph is returned ($FRAME$).

$$\frac{Scenario_i \to ABS_i \quad ABS = \bigcup_i ABS_i}{Scenario* \to \{ABS\}} \qquad SB$$

$$\frac{\begin{array}{c} Env \to \sigma, Def \quad \sigma \vdash Frame_i \to G_i, Label_i \\ G = \bigcup_i G_i \quad ABS = \{\mathsf{Preds}(G), S(G), \bigcup_i Label_i \to S(G_i), Def\} \end{array}}{Env\ Frame* \to ABS} \qquad SCEN$$

$$\frac{\sigma \vdash Constraint \to G}{\sigma \vdash Label\ Constraint \to G, Label} \qquad FRAME$$

$$\frac{\sigma \vdash Pred_i \to (u_i, v_i) \quad E = \bigcup_i \{(u_i, v_i)\} \quad V = \bigcup_i \{u_i, v_i\}}{\sigma \vdash Pred* \to (V, E)} \qquad CONSTR$$

$$\frac{Pred = (u = v)}{\sigma \vdash Pred \to (\sigma(u), \sigma(v))} \qquad PRED$$

Figure 4-3: Semantics of the Storyboard language

Chapter 5

EXPSYN Algorithm

In this chapter, we explain how the EXPSYN algorithm synthesizes the desired implementation from the control flow sketch and the storyboards. First, the data flow constraints are obtained from the control flow sketch using a fairly standard technique [9]. Then an uninterpreted relation mapping an abstract state to a set of abstract states is constructed for each of the transition function candidate present in the control flow sketch. This relation is then lifted to a function mapping an abstract set-state to another abstract set-state. Finally the function choice at each program point in the data flow constraints are encoded in the high level SKETCH intermediate language. The novelty of this algorithm lies in the way we construct these transition functions from the abstract domain using abstract interpretation [7] and the high level encoding of the function choices for every statement. We describe each of these steps in more details.

5.1 Notations

From chapter 4, we derive a set of abstract states for every scenario in a storyboard and we denote this set by $\mathcal{S}$. Let the size of this set $|\mathcal{S}|$ be N. We add an error state s_{err} to this set to get $\mathcal{S}' = \mathcal{S} \cup \{s_{err}\}$. The transition function candidates in the control flow sketch are denoted by τ and their corresponding uninterpreted functions are represented as f_τ. The uninterpreted functions in the data flow equations are represented as f_i. We represent a set of states at program point i as a bitvector t_i of size $N + 1$, where the j^{th} bit of the bitvector $t_i[j]$ represents the reachability of the state $s_j \in \mathcal{S}'$ at location i. The sketch control choice variables $c_{??}$ represent holes that can take any integer value.

5.2 Construction of uninterpreted transition functions

Algorithm 2 describes the construction of an uninterpreted transition relation f_τ for a transition function τ. The algorithm iterates through all pairs of states $(s, s') \in \mathcal{S}^2$ such that $s \Longrightarrow \mathsf{WP}(s', \tau)$, i.e. state s implies the constraint obtained from weakest precondition of state s' on transition τ. The weakest precondition implication query is answered by an off the shelf CLP prolog theorem prover [31]. All such state pairs (s, s') are added to the transition relation f_τ. It is to be noted that an abstract state s can map to multiple abstract states s' on a transition relation f_τ due to the nondeterminism introduced by the abstraction. For all statement transitions τ (ignoring conditional transitions), the transition relation f_τ is completed with (s, s_{err}) for all states $s \in \mathcal{S}$ if there exists no $s' \in \mathcal{S}$ such that $(s, s') \in f_\tau$. Also (s_{err}, s_{err}) is added to f_τ (once the program reaches an *error* state, it stays in an error state).

Algorithm 2 BuildTransitions(τ)

Require: the transition τ, $\mathcal{S}$: set of abstract states
Ensure: the transition relation $f_\tau : \mathcal{S} \rightarrow 2^{\mathcal{S}}$

```
1: for s in S do
2:   for s' in S do
3:     if s ⟹ WP(s', τ) ∧ ScenarioId_s = ScenarioId_s' then
4:       add (s, s') to f_τ
5:     end if
6:   end for
7: end for
```

Algorithm 3 $\mathsf{fc}_\tau(f_\tau, t_{in})$

Require: f_τ : the uninterpreted transition relation, t_{in}: input abstract states bitvector
Ensure: t_{out} : bitvector of output abstract states

```
1: t_out = 0
2: for i = 0 to N do
3:   if t_in[i] == 1 then
4:     for (s_i, s) ∈ f_τ do
5:       t_out[s] = 1
6:     end for
7:   end if
8: end for
```

The above constructed uninterpreted functions f_τ are then lifted to corresponding functions fc_τ that map an abstract set-state to another abstract set-state. Algorithm 3 describes how the transition functions fc_τ are encoded in the SKETCH intermediate language. For every set bit i in the input bitvector t_{in}, it sets the

corresponding bit j in the output bitvector t_{out} where $(s_i, s_j) \in f_\tau$. The algorithm constructs the function fc_τ implicitly as it is expensive (in some cases impossible) to enumerate all possible input bitvectors of size N.

5.3 SKETCH encoding of data flow constraints

The data flow constraints obtained from the control flow sketch of the program are first translated into the SKETCH language. The translation for the example data flow sketch in section 2.3.1 is shown below. The initial and final state constraints on the bitvectors are asserted using setInitialState and checkFinalState methods respectively.

```
void main(){
  bit[N] t_in, t_1, t_2, t_3, t_out;
  int c_1, c_2, c_3, c_4, c_5;

  c_1 =??; c_2 =??; c_3 =??; c_4 =??; c_5 =??;

  t_in = setInitialState();

  t_1 = f_1(t_in, c_1)|f_2(t_2, c_2);
  t_2 = f_c(t_1, c_3);
  t_3 = f_c̄(t_1, c_4);
  t_out = f_3(t_3, c_5);

  assert checkFinalStates(t_out);
  }
```

Each of the data flow constraint function f_i is encoded in the SKETCH intermediate language as follows. The choice parameter value delegates the function call to the corresponding transition function relation fc_c indexed by the choice variable.

```
bit[N] f_i(bit[N] t, int c){
  if(c == 1) return fc_1(t);
  if(c == 2) return fc_2(t);
  if(c == 3) return fc_3(t);
  if(c == 4) return fc_4(t);
  ⋮
}
```

After encoding the constraints in the SKETCH language, the SKETCH solver is used to search for a satisfying assignment to the choice parameters $c_{??}$ which are then mapped back to the corresponding statement functions in the original control flow sketch of the program.

5.4 Correctness guarantees of the synthesized implementation

We only get partial correctness guarantees about the synthesized implementation from this algorithm. We do not guarantee termination as the solution to the set of data flow equations is not necessarily a least fixed point solution. If the program terminates on an input, then it is guaranteed to return the correct result.

Chapter 6

IMPSYN Algorithm

The EXPSYN algorithm presented in chapter 5 works well for storyboards with upto 300 abstract states approximately. Many of the manipulations require much larger number of abstract states, e.g. the binary search tree rotation manipulation storyboard gives rise to more than 1000 abstract states. The IMPSYN algorithm is developed to handle such large abstract state space manipulations. The algorithm maintains the abstract states and the transition functions symbolically and only constructs abstract states which are reachable during the execution. The main idea this algorithm exploits is the fact that most of the abstract states are not reachable during the program execution; and they need not be computed if they are not required by the synthesizer. This representation trades time complexity with the memory complexity, as the solver now has to symbolically execute the transitions itself but the memory requirements are significantly reduced.

The EXPSYN algorithm performs synthesis and verification of the implementation both together simultaneously. It can be noted that the verification of the algorithm is a completely deterministic process and the solver's expensive search for the verification purpose can be avoided if handled separately. The IMPSYN algorithm consists of separate synthesis and verification engines which communicate through the counterexample traces; thereby reducing the burden on the synthesis solver to only search for satisfying the constraints on path program traces [2] instead of arbitrary control flow. We name it counterexample trace guided inductive synthesis (CETGIS) and the detailed description of the algorithm follows.

6.1 Symbolic representation of states and transition functions

The abstract states obtained from the storyboards are represented symoblically using an `astate` type of type `struct` in SKETCH. All the concrete and abstract locations in the storyboard are defined as integer macros. Each class variable in the storyboard is represented as an integer field in `astate`. The updatable pointer locations are represented as two dimensional integer arrays in `astate` where the first index represents the location and the second index represents the nondeterministic choice introduced by

the abstraction. These integer variables can take any value from the integer macros defined for the abstract and concrete locations. Additionally astate also contains two bits isErr for representing whether the abstract state is an error state and isEmpty for representing whether a state is an empty state.

The abstract state representation of the storyboard in section 2.1.3 is shown below. The abstract and concrete node locations FRONT, A, X, B, BACK and null are defined as integer macros. The struct astate contains the variables head, prev and curr with the two-dimensional array next corresponding to the updatable pointer locations. This example has a non-determinism branching factor of 2. e.g. next[FRONT][0] stores the first nondeterministic choice for front.next and next[FRONT][1] stores the second nondeterministic choice for front.next.

```
#define FRONT 0
#define A 1
#define X 2
#define B 3
#define BACK 4
#define EMP 5 // Null

struct astate{
  int head, prev, curr;
  int[5][2] next;
  bit isErr;
  bit isEmpty;
}
```

The transition functions are similarily represented symbolically. A transition function fc_i takes as an argument an input abstract state and an integer choice variable c representing how to resolve the nondeterminism, and returns an accordingly modified output abstract state. The transition functions also model error conditions if a null pointer is dereferenced. The transition functions for the conditional statements sets the isEmpty bit according to the conditional expression.

An example symbolic transition function for the example in section 2.1.3 is shown below.

```
  // curr = curr.next
astate fc12(astate fromState, bit c){
  astate toState = new astate(fromState);

  if(toState.curr == EMP)
    fromState.err = 1;
  else
    toState.curr = fromState.next[c][fstate];

  return toState;
}
```

6.2 SKETCH encoding and constraints

The sketch encoding for the data flow constraints consists of four kinds of constraints:

- **Program trace constraints (Trace):** These traces correspond to an execution path in the control flow sketch of the program. The path program traces consists of a sequence of program statements in the control flow sketch starting from the start location and ending at the return location. It is to be noted that these program traces are completely deterministic as the non-deterministic choice is resolved by the concrete value of the second parameter. An example program trace constraint for the running example is shown below.

```
void trace1() implements spec{
  astate[10] t;
  t[0] = getInitialState();

  t[1] = f1(t[0], 0);
  t[2] = fc(t[1], 0);
  t[3] = f2(t[2], 1);
  t[4] = fc(t[3], 0);
  t[5] = f2(t[4], 1);
  t[6] = fc(t[5], 0);
  t[7] = f2(t[6], 1);
  t[8] = fc̄(t[7], 0);
  t[9] = f3(t[8], 0);

  assertFinalState(t[9]);
}
```

- **Loop exiting constraints (Termination):** For every loop in the control flow sketch, these constraints encode the constraint that there exists at least one state in some nondeterministic trace execution that exits the loop after a bounded number of loop unrollings. The holes ?? can take any bit value to construct some nondeterministic behaviour. These constraints ensure the terimnation of the synthesized implementation in presence of the loops. An example constraint for the example with loop bound of $k = 4$ is presented below.

```
void loop1Exit() implements spec{
  astate[10] t;
  t[0] = getInitialState();

  t[1] = f1(t[0], ??);

  t[2] = fc(t[1], ??);
  t[3] = f2(t[2], ??);

  t[4] = fc(t[3], ??);
  t[5] = f2(t[4], ??);
```

```
    t[6] = fc(t[5], ??);
    t[7] = f2(t[6], ??);

    t[8] = fc(t[7], ??);
    t[9] = f2(t[8], ??);

    assert t[2].isEmpty || t[4].isEmpty || t[6].isEmpty || t[8].isEmpty;
}
```

- **Loop fixpoint constraints (Fixpoint):** For every loop in the control flow sketch, these constraints encode the constraint that the loop computation reaches a fixpoint after a bounded number of loop unrollings. An example constraint for the example is shown below with the bound for loop unrolling $k = 4$.

```
void loop1Fixpoint(bit[9] c) implements spec{
  astate[10] t;
  t[0] = getInitialState();

  t[1] = f1(t[0], c[0]);

  t[2] = fc(t[1], c[1]);
  t[3] = f2(t[2], c[2]);

  t[4] = fc(t[3], c[3]);
  t[5] = f2(t[4], c[4]);

  t[6] = fc(t[5], c[5]);
  t[7] = f2(t[6], c[6]);

  t[8] = fc(t[7], c[7]);
  t[9] = f2(t[8], c[8]);

  assert equal(t[2],t[4]) || equal(t[2],t[6]) || equal(t[2],t[8])
       || equal(t[4],t[6]) || equal(t[4],t[8]) || equal(t[6],t[8]);
}
```

- **Final state constraint (FinalState):** The final state constraint (assertFinalState(t)) encodes the constraint that the final state is either an empty state (t.isEmpty = 1) or the final state satisfies the constraints specified by the End frame constraints of the storyboard.

The uninterpreted function choices in the data flow constraints are encoded similary as in Section 5.3 and is shown below.

```
bit[N] fi(bit[N] t, bit c){
  if(??) return fc1(t, c);
  if(??) return fc2(t, c);
  if(??) return fc3(t, c);
  if(??) return fc4(t, c);
  ⋮
}
```

At every program location, the sketch synthesizer needs to maintain only one abstract state. Each trace is sequentially added to the constraints and are solved incrementally exploiting the incremental solving feature of SKETCH. The incremental solving feature is essential for feasibility of solving the large set of constraints.

6.3 Verification engine

The verification engine implements a standard deterministic abstract fixpoint verification algorithm as shown in algorithm 4. The algorithm computes the set of all reachable states of the input program until a fixpoint of abstract states is reached. If the final state constraints are satisfied, the verifier returns that the program P is correct. Otherwise, the verifier returns a deterministic counterexample trace (with concrete choice values) back to the synthesis engine.

Algorithm 4 Verify(P)

Require: P : program implementation with concrete transition functions
Ensure: a deterministic counterexample trace cex if not correct, otherwise null

```
1: Queue<astate> stateQueue = empty
2: HashSet<astate> visitedStates = empty
3: stateQueue.enqueue(initState)
4: while stateQueue != empty do
5:    astate s = stateQueue.dequeue()
6:    visitedStates.add(s)
7:    for astate s' ∈ nextStates(s) do
8:       if s'∉ visitedStates then
9:          stateQueue.enqueue(s')
10:      end if
11:   end for
12: end while
13: if checkFinalState(visitedStates) then
14:   return null { /* Program is verified */}
15: else
16:   return getCexTrace(visitedStates)
17: end if
```

6.4 IMPSYN algorithm description

The IMPSYN algorithm communicates between the synthesis engine and the verification engine as shown in figure 6-1. The detailed algorithm is shown in Algorithm 5. First the verifier assigns a random assignment to the uninterpreted transition functions in the data flow constraints and then runs the verifier on that program. If the program verifies, the current program is returned as the desired data structure manipulation implementation. Otherwise a deterministic counterexample path from the

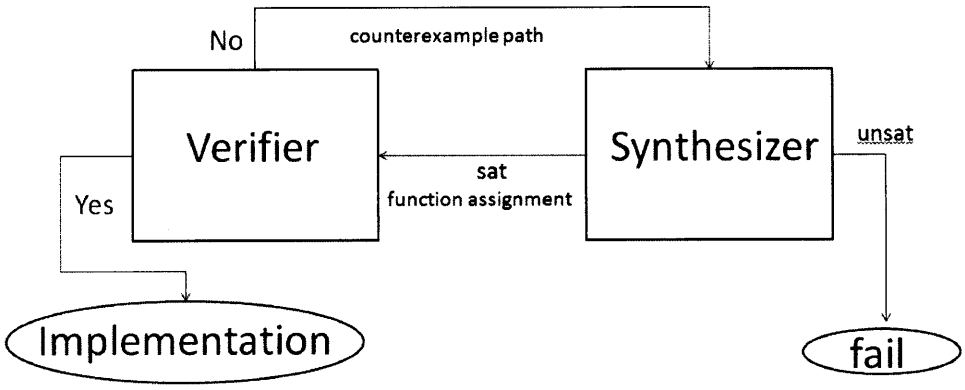

Figure 6-1: The IMPSYN algorithm

start location to the return location of the program is returned to the synthesis engine. The synthesizer incrementally adds the counterexample trace constraints to the loop exit, loop fixpoint and final state constraints. It then searches for a satisfying assignment to the functions to remove this counterexample path. The synthesized program is then returned back to the verifier for verifying whether the currently synthesized program is correct. If yes, the algorithm stops and the synthesized implementation is returned. Otherwise the loop between the synthesis and verifier engines continues.

Algorithm 5 IMPSYN

Require: the data flow constraints and correctness conditions from storyboard
Ensure: the correct data structure implementation

```
1: Assign function randomly in P
2: boolean done = false
3: while !done do
4:    cex = Verify(P)
5:    if cex == null then
6:       done = true
7:    end if
8:    P = Synthesize(P, cex)
9: end while
10: return P {The desired correct implementation}
```

6.5 Correctness proof of IMPSYN algorithm

In this section, we provide a brief proof about the correctness of IMPSYN algorithm, i.e. a correct program P is returned by the IMPSYN algorithm. For simplicity and without loss of generality let us assume we have only a single scenario with one

input state constraint (s_{in}) and one output state constraint (s_{out}). The final state constraint specifies that at the end of the trace the output state can either be empty or the specified output state (s_{out}). All the states reachable at the return location of the program are guaranteed to satisfy the output state constraints (FinalState constraint). Therefore for proving correctness of the algorithm, it suffices to show that there exists at least one non-empty state reachable at the return location of the program. We need to consider the three kinds of basic control flow constructs present in the program :

- **Straight Line :** If the initial state is non empty (initState.isEmpty = 0), then the final state is also going to be non empty as the only way to make a state empty is through some conditional statement.
- **Conditional Branch :** Let us call the two branches of the conditional branches to be b_{true} and b_{false}. For contradiction, let us assume both the branch paths leads to an empty final state. For this condition to happen, the initial state should not pass both the true and the false branch of the conditional cond, i.e. the state satisfies both cond and !cond which is a contradiction. Therefore at least one of the paths lead to the correct final state.
- **Loop :** With the loop exit constraint we can guarantee that there exists at least one abstract state that leaves the exits the loop within the bound of the loop unrolling.

These correctness arguments can be naturally extended inductively to an arbitrary combination of the these building blocks to obtain correctness arugments for programs with arbitrary control flow.

6.6 Optimizations

In this section, we present some optimizations that accelerates the loop between the Verifier and Synthesizer engines to obtain faster convergence.

6.6.1 Synthesizing multiple program paths together

The Verifier returns a completely deterministic path program trace to the synthesizer phase. Every path program statement has a non-deterministic choice parameter. Let the non-determinism bound be b and number of statements in the trace be n. There are n^b possible traces with the same program statements and the Verify-Synthesis loop might take a long time in correcting multiple similar paths. As an optimization step, the Synthesizer treats the concrete choices of the deterministic counterexample path trace as an input variable to the program. The SKETCH system synthesizes a program that satisfies the final state constraints for all possible valuations of the choice parameters, i.e it satisfies all n^b possible paths simultaneously. SKETCH uses CEGIS algorithm to automatically discover only the required non-deterministic paths from

the set of all paths. For the example constraint in section 6.2, the trace is updated as shown below.

```
void trace1(bit[9] c) implements spec{
  astate[10] t;
  t[0] = getInitialState();

  t[1] = f1(t[0], c[0]);
  t[2] = fc(t[1], c[1]);
  t[3] = f2(t[2], c[2]);
  t[4] = fc(t[3], c[3]);
  t[5] = f2(t[4], c[4]);
  t[6] = fc(t[5], c[5]);
  t[7] = f2(t[6], c[6]);
  t[8] = fc̄(t[7], c[7]);
  t[9] = f3(t[8], c[8]);

  assertFinalState(t[9]);
}
```

6.6.2 Merging conditional statements in path program traces

The conditional statements can similarily create exponential number of paths even without a non-deterministic choice. Therefore the conditional statements are treated as a part of the path program traces. The idea we exploit here is the fact that even in the presence of conditional statements in the program trace, the synthesizer still needs to keep track of only 1 program state at each program location. This is because at merge points only 1 path is feasible.

6.6.3 Concretizing the choices for loop exit constraint

The loop exit constraint states the constraint about existence of at least one state exiting the loop condition after a bounded number of loop unrolling. Since the final state constraint states that either the final state can be empty or it satisfies the end frame constraint, it is much easier for synthesizer to come up with a satisfying assignment that leads the final state to an empty state. There is an existential quantification over the non-deterministic choices in the program statements in the loop exit constraint. Therefore given some number of deterministic counterexample traces, the synthesizer can fill up the existential holes in the loop exit constraint to make the final states empty in a much easier fashion. For most of the inductive cases, we reserve the 1^{st} non-deterministic choice for the base case and other ones for inductive cases. We want the loop exit constraint to hold on the base case and the inductive case is only used for reaching a fixpoint. We apply this observation as an optimization by simply replacing the non-deterministic holes with a concrete value 0. The loop exit constraint section 6.2, the trace is updated as shown below.

```
void loop1Exit() implements spec{
  astate[10] t;
  t[0] = getInitialState();

  t[1] = f1(t[0], 0);

  t[2] = fc(t[1], 0);
  t[3] = f2(t[2], 0);

  t[4] = fc(t[3], 0);
  t[5] = f2(t[4], 0);

  t[6] = fc(t[5], 0);
  t[7] = f2(t[6], 0);

  t[8] = fc(t[7], 0);
  t[9] = f2(t[8], 0);

  assert t[2].isEmpty || t[4].isEmpty || t[6].isEmpty || t[8].isEmpty;
}
```

6.7 Correctness guarantees of the synthesized implementation

This algorithm provides complete correctness guarantees for the synthesized implementation, as it also provides a termination guarantee in addition to the final states satisfying the end frame constraints. This algorithm requires a bound on the number of loop unrollings k to reach a fixpoint, but in most cases it is observed that this number is small in the abstract setting. This value k can be varied algorithmically as well if required.

Chapter 7

Experiments

This chapter presents preliminary results for the EXPSYN and IMPSYN algorithms on some data structure manipulation case studies including linked lists and binary search trees. The experiments were run on an Intel Core 2 Quad 3.0 GHz CPU with 4GB of RAM.

7.1 EXPSYN algorithm results

The results for case studies of EXPSYN algorithm are presented in Table 7.1. The Table describes the number of clauses in the corresponding SAT formula, the amount of memory used, the number of abstract states, the number of transition functions and the time it took to synthesize the desired implementation. The state column denotes the number of abstract states obtained from the storyboard after the state edge cover optimization. The transitions corresponds to the number of possible choices for a program statement in the control flow sketch. From the Table, we can observe that the EXPSYN algorithm is very memory intensive. Since it encodes the abstract set-states as a bitvector, the search space over bitvectors of such large sizes results in large memory requirements. A brief description for the case studies, control flow sketch and the synthesized implementations follow.

Data Structure	#Clauses	Memory	States	Transitions	User Time
Link list insertion	179K	5.5G	249	73	6m8s
Link list deletion	180K	5.5G	249	73	5m46s
BST insertion	258K	8.1G	211	78	2m32s

Table 7.1: Experimental results for EXPSYN algorithm

7.1.1 Insertion/Deletion in a sorted linked list

We first desrcibe the results on our running example. The storyboard for the sorted linked list insertion manipulation is shown in figure 7-1. The four scenarios correspond to: insertion in the middle of the linked list, inserting in the beginning of a linked

list, inserting in the end of a linked list and inserting in an empty linked list. The control flow sketch of the program is shown in figure 7.1.1.

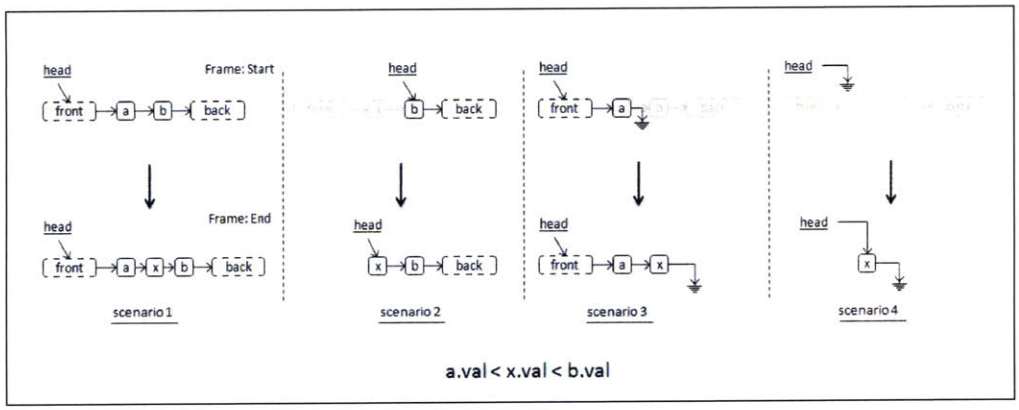

Figure 7-1: Storyboard for linked list insertion

The storyboard for the sorted linked list deletion manipulation is the storyboard obtained by flippping the input and output frames of the storyboard for the sorted linked list insertion manipulation shown in figure 7-1. The control flow sketch of the deletion operation was also given the same as the insertion opertation shown in figure 7.1.1.

It is interesting to note that for the reversible manipulations, we can use this trick of swapping the input and output frame constraints to achieve the inverted manipulation. Another example of such invertible operation is left rotation and right roation in a binary search tree.

It is interesting to note that for this example, we did not require to specify the inductive definiton of the abstract nodes front and back. These definitons were not required for this case as the abstract nodes are only required to perform scanning and reaching the correct concrete nodes for local modifications. For the scan and modify manipulations, the inductive definitions would both perform the same execution and would not add any new behaviour to the program execution. The storyboard specification without the inductive definitions might allow some bad inputs to be abstracted away as the abstract input specification, e.g. the abstract node front can represent a set of nodes with an internal cycle. But the synthesized implementation would still provide the partial correctness guarantee, i.e it would either not terminate or terminate with the correct result. Adding inductive definitions to the storyboard would rule out such bad inputs.

```
#define LOC {| (head|prev|curr|x)(.next)? |}
#define STMT LOC = LOC;
#define COND LOC == null
#define LCOND LOC!=null && LOC.val < x.val

void llInsert(Node head, Node x){
/*@Start*/
    Node prev, curr;
    STMT
    STMT
    while(LCOND){
        STMT
        STMT
    }
    if(COND){
      STMT
      STMT
    }
    else{
      STMT
      STMT
    }
/*@End*/
}
```

Figure 7-2: Control flow sketch for sorted linked list insertion manipulation

```
void llInsert(Node head, Node x){

    Node prev=null, curr=null;
    x.next = prev;
    curr = head;

    while(curr!=null && curr.val < x.val){
        prev = curr;
        curr = prev.next;
    }

    if(curr == head){
      x.next = curr;
      head = x;
    }
    else{
      x.next = curr;
      prev.next = x;
    }

}
```

Figure 7-3: Synthesized implementation for sorted linked list insertion manipulation

```
void llDelete(Node head, Node x){

    Node prev=null, curr=null;
    curr = x;
    curr = head;

    while(curr!= x){
        prev = curr;
        curr = curr.next;
    }

    if(curr == head){
      head = head;
      head = head.next;
    }
    else{
      prev.next = curr.next;
      prev.next = prev.next;
    }

}
```

Figure 7-4: Synthesized implementation for sorted linked list deletion manipulation

7.1.2 Insertion in a binary search tree

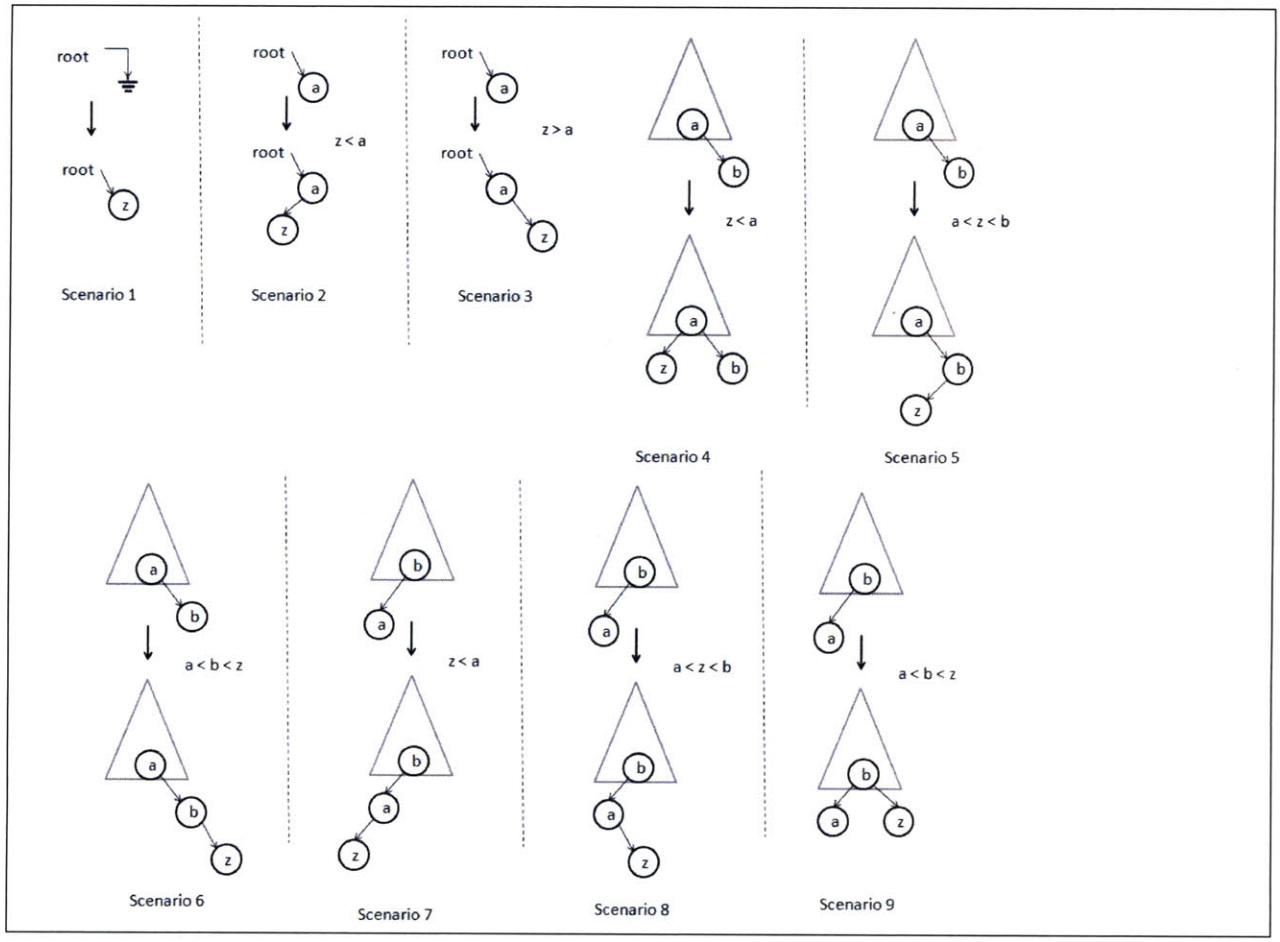

Figure 7-5: Storyboard for binary search tree insertion

The storyboard for inserting a node z in a binary search tree is shown in figure 7-5. The storyboard consists of 9 scenarios: inserting a node in an empty tree, two scenarios for inserting a node in a tree with one node (left and right), three scenarios for inserting a node with a.right = b and z < a, a < z < b and b < z and three scenarios with b.left = a and z < a, a < z < b and b < z. The control flow sketch of the bst insertion operation is shown in figure 7.1.2. The control flow structure has a while loop as it is expected to be an $O(n)$ manipulation. The loop body contains an assignment statement and a conditional statement. After the loop, the sketch contains two nested conditional statements. It can be seen that in this case, there is a lot more structure which is required to be provided by the programmer. But we want to emphasize that this part of exploring different control flow structures can be automated as an outer loop to our current framework. Performing a naive exploration would certainly be very expensive, but an exploration guided by programmer insight would be much more useful and general.

The storyboard again for this case study does not include the inductive definitions for the the abstract nodes. It can allow some ill-formed input to be abstracted away as the abstract specification. But the combination of different scenario cases and the control flow sketch restricts the possibility of insertion at some incorrect position in the tree. We believe that for scan and modify manipulations, we can get away with some incomplete specifications which are not essential for synthesizing the correct implementations.

```
#define LOC {| (x|y|z|root)(.left | .right)? |}
#define STMT LOC = LOC;
#define LCOND {| (x | y)(.left | .right)? != null |}
#define NCOND {| (x|y|z).key < (x|y|z).key |}
#define COND {| LCOND | NCOND |}

void bstInsert(Node head, Node x){
/*@Start*/
   Node prev, curr;
   STMT
   STMT
   while(COND){
      STMT
      if(COND)
        STMT
      else
        STMT
   }
   if(COND){
     STMT
   }
   else{
     if(COND)
       STMT
     else
       STMT
   }
/*@End*/
}
```

Figure 7-6: Control flow sketch for binary search tree insertion manipulation

```
void bstInsert(Node root, Node z){

    Node x = null, y = null;
    x = root;
    x = root;

    while(x != null){
        y = x;
        if(x > z)
            x = y.left;
        else
            x = y.right;
    }

    if(y != null){
      if(y > z)
            y.left = z;
      else
            y.right = z;
    }
    else
        root = z;

}
```

Figure 7-7: Synthesized implementation for binary search tree insertion manipulation

7.2 IMPSYN algorithm results

The results for IMPSYN algorithm on BST rotation and linked list reversal case studies is presented in Table 7.2. It can be observed from the Table that the memory requirements for IMPSYN algorithm are significantly lesser than that for the EXPSYN algorithm. The number of abstract states in these benchmarks are much larger than the threshold ($\approx$ 300 states) for the EXPSYN algorithm. It can also be noted that the sizes of the SAT formula is much larger. The incremental solving feature of the IMPSYN algorithm enables the solving of such large SAT instances.

Data Structure	#Clauses	Memory	States	Transitions	Iterations	Total Time
BST left-rotation	1.36M	0.8G	9^4	79	1	113s
BST right-rotation	1.35M	0.7G	9^4	79	1	115s
Linked List reversal	611K	0.5G	11^4	76	4	110s

Table 7.2: Experimental results for IMPSYN algorithm

7.2.1 Left/Right rotation in binary search tree

The storyboard for the BST left rotate manipulation is shown in Figure 7-8. This is taken directly from the Introduction to Algorithms textbook [6]. The first scenario shows the common case with two concrete nodes x and y which are to be appropriately rotated. The abstract nodes α, β, γ and PX represent a set of nodes in the corresponding subtree. As the book also points out, the algorithm needs to care about whether the node x is the root node or x is left/right child of its parent; and if the left subtree of node y is empty. This gives us 6 scenarios in total as shown in the storyboard.

The control flow sketch is shown in Figure 7.2.1. Here STMT is defined to be all statements over the variables root, x and y with two pointer dereferences. Since the storyboard has 6 scenarios and the expected complexity of the manipulation is $O(1)$, the programmer provides three if statements. The second nested if corresponds to the case that there are three cases for parent pointer of x : null, parent to the left and parent to the right. It can be noted that this structure of control flow sketch can be easily automated as well. For this example, we can try different combinations of if statements nesting. The synthesized implementation for left rotation is presented in Figure 7.2.1.

As in the case of the linked list deletion previously, for synthesizing the BST right rotation implementation we simply reverse the input output pairs in the scenarios of the storyboard in Figure 7-8. The control flow sketch essentially remains the same with conditionals choices flipping symmetrically. The synthesized code is shown in Figure 7.2.1.

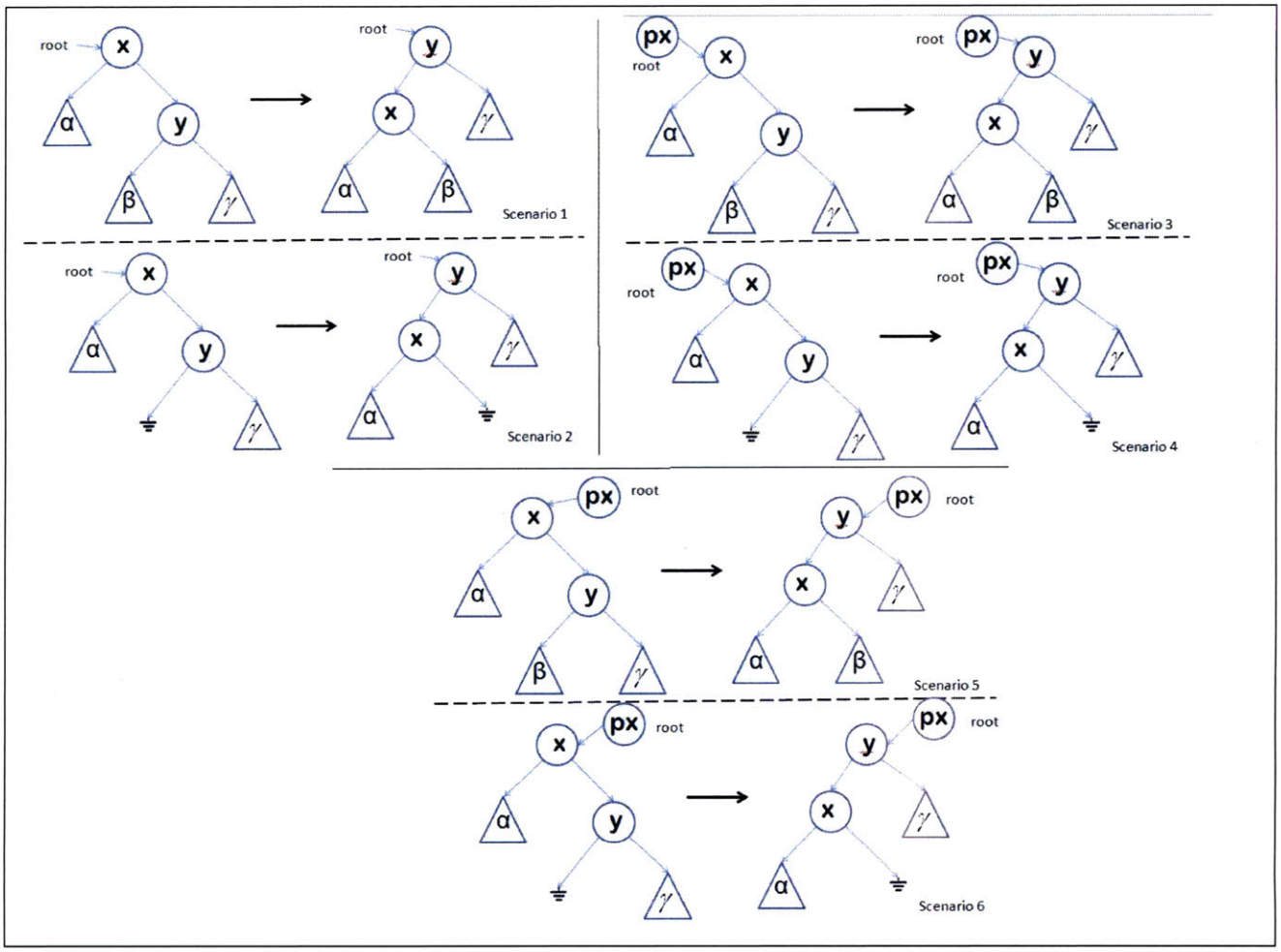

Figure 7-8: Storyboard for binary search tree left rotation

```
#define LOC {| (x|y|root)(.left | .right | .parent)?(.left | .right | .parent)? |}
#define STMT LOC = LOC;

#define COND {| y.left == null | x.parent == null | x == x.parent.left |}

void bstLeftRotate(Node root, Node x, Node y){
/*@Start*/
    STMT
    STMT
    if(COND){
      STMT
    }
    else{
      STMT
    }
    STMT
    STMT
    if(COND){
       if(COND)
         STMT
       else
         STMT
    else
      STMT
    }
    STMT
    STMT
/*@End*/
}
```

Figure 7-9: Control flow sketch for binary search tree left rotation manipulation

```
void bstLeftRotate(Node root, Node x, Node y){

  root = root.parent;
  y.parent.right = y.left;

  if(y.left != null)
     y.left.parent = y.parent;

  y.left = x.left.parent;

  if(x.parent == null)
    root = y.right.parent;
  else{
    if(x == x.parent.left)
     root.parent.left = y.right.parent;
    else
     root.parent.right = y.right.parent;
  }

  y.parent = y.left.parent;
  x.parent = y.right.parent;

}
```

Figure 7-10: Synthesized implementation for binary search tree left rotate manipulation

```
void bstRightRotate(Node root, Node x, Node y){

  y.left = y.left.right;
  root = root.parent;

  if(x.right != null)
     x.right.parent = y;

  x.right = y.right.parent;

  if(y.parent == null)
    root = x.left.parent;
  else{
    if(y == y.parent.left)
     root.parent.left = x.left.parent;
    else
     root.parent.right = x;
  }

  x.parent = root.parent.parent;
  y.parent = x;

}
```

Figure 7-11: Synthesized implementation for binary search tree right rotate manipulation

7.2.2 In-place linked list reversal

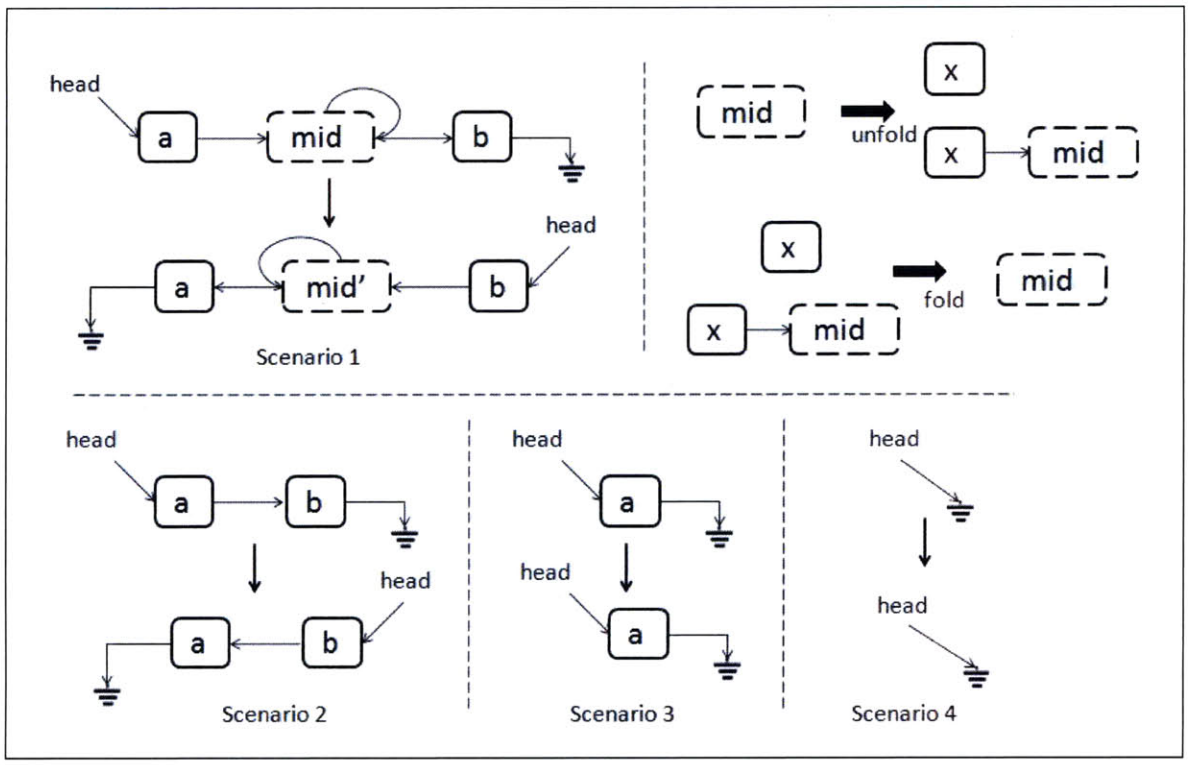

Figure 7-12: Storyboard for inplace linked list reversal

The storyboard for inplace linked list reversal is shown in Figure 7-12. The first scenario defines the common case where the first node of list is defined to be a concrete node a, the last node as concrete node b and all the nodes in the middle between these two nodes are represented by the abstract node mid. The inductive definition of the abstract node mid is also shown in the storyboard by the nondeterministic unfold and fold operations. The other scenarios cover the cases of linked lists of length 2, 1 and 0 respectively.

The control flow sketch of the manipulation is shown in Figure 7.2.2. The programmer provides the expected complexity of $O(n)$ with one while loop with randomly chosed 5 statments. This number of statements inside the loop can be algorithmically manipulated. The programmer also expects that three temporary variables temp1, temp2 and temp3 would be sufficient. Every loop iteration begins with an unfold statement and ends with a fold statement. The synthesized implementation is shown in figure 7.2.2. Interestingly, it can be observed that the synthesized implementation does not use the temporary variable temp2. Also, only 4 statements are required inside the loop body.

```
#define LOC {| (head | temp1 | temp2 | temp3)(.next)? |}
#define STMT LOC = LOC;
#define COND {| temp1 != null | temp2 != null | temp3 !=null | head != null |}
#define UNFOLDSTMT {| unfold LOC |}
#define FOLDSTMT {| fold LOC |}

void llReverse(Node head){
/*@Start*/

  Node temp1=null, temp2=null, temp3=null;

   STMT

   while(COND){
     UNFOLDSTMT
     STMT
     STMT
     STMT
     STMT
     STMT
     FOLDSTMT
   }

/*@End*/
}
```

Figure 7-13: Control flow sketch for inplace linked list reversal manipulation

```
void llReverse(Node head){
  Node temp1 = null, temp2 = null, temp3 = null;

  temp1 = head;

  while(temp1 != null){
    // unfold temp1;
    head = temp1;
    temp1 = temp1.next;
    head.next = head;
    head.next = temp3;
    temp3 = head;
    // fold head;
  }

}
```

Figure 7-14: Synthesized implementation for inplace linked list reversal manipulation

Chapter 8

Related Work

8.1 Software synthesis

Software synthesis has been an active area of research at least since the early 80s when the seminal work of Waldinger and Manna [15, 14] showed some early promise in the concept of deductive synthesis. A more algorithmic approach to synthesis was pioneered by Pnueli and Rosner in the context of finite state controllers [19]. More recently, some of the ideas from the field of controller synthesis have been applied to software, for example, to synthesize program repairs [11].

The idea of using abstract interpretation for synthesis was recently introduced by Vechev Yahav and Yorsh [30], as a follow up to earlier work on synthesis of concurrent datastructures [29]. Their system is designed to synthesize efficient synchronization for concurrent programs, and is very different from ours, both in its scope and in the algorithms it uses. Unlike our system, their synthesizer is not based on constraint solving; instead, it uses a "generate-and-test" approach, where the system generates candidate implementations and tries to verify them using abstract interpretation, relying on the domain-specific properties of the domain to prune the search space. This approach is very effective for the problem of synthesizing synchronization, but the constraint based approach is more general and allows us to handle extremely large search spaces with no apparent structure.

The idea of using a constraint based approach for abstract interpretation was previously introduced by Gulwani et. al. [9]. More recently, their group has used similar techniques to synthesize invariants [10] and even complete programs [27]. The most important distinction between their work and ours is the use of the storyboards to capture insights from the programmer and make the synthesis process more efficient.

The idea of using a sketch to define the structure of the implementation was adapted from the original work on sketch based synthesis [24]. The idea was originally applied to the domain of bit-stream manipulations [26], such as ciphers and error correction codes, and has been applied more recently to scientific programs [24] and concurrent datastructures [25].

8.2 Visual Programming and Programming by Example

The systems using graphics for aiding programming, debugging and program understanding have been an intriguing research area from a very long time. Myers [17] classifies these systems on the basis of three broad categories: Visual Programming (with Program Visualizations), Programming by Example and interactive/batch systems. Visual Programming refers to systems that allow programmers to specify program computations graphically whereas Program visualization is used for graphically visualizing data structures at run-time for debugging purposes. Programming by Example approach uses a finite set of input-output pairs and tries to infer a program that conforms to those examples.

Grail [8] was one of the earliest systems that compiled flowcharts to executable code. The AMBIT/G [5] language represented both programs and the data as graphs. Then the pictorial program was pattern matched for its execution. The framework was used to describe list-structure garbage collection program and reduction-analysis string parser. Even though these approaches alleviate the problem of writing code by letting the programmers use static predefined pictures but the burden of figuring out the exact sequence of operations still lies with the programmer. Also attempting to capture dynamic transformations trhough static diagrams makes the resulting programs much difficult.

Shaw [21] developed a framework for learning restricted Lisp programs from single input/output. The framework is not for general programs and also not guaranteed to learn the correct program. Pygmalion [22] was one of the first successful programming by demonstration systems. The programmer provided concrete execution of the program on a concrete example with the help of icons and the system inferred some recursive program from the example. Tinker [13], aimed at beginning programmers, let them write Lisp programs by providing Lisp expressions or mouse inputs to handle the execution on concrete examples of input data. These concrete program executions were then generalized to symbolic executions and in the process ambiguities were resolved by asking the programmer for disambiguations. These systems alleviates the problem for programmer to worry about abstract inputs but still they require the programmer to know how the program is supposed to execute on concrete inputs.

Sketchpad [28] is a seminal work that lead to a whole new field of human-computer interaction, as well is considered a great breakthrough for the computer graphics research. In this framework, a programmer used sketched geometrical object shapes like straight lines, arcs etc. using a *light pen*. The programmer could also express constraints on these shapes to get regular geometrical objects on the screen and used graphical buttons for providing options like copying etc. Even though this was a revolutionary work, this did not cater to the general purpose programming puroposes.

Thinkpad [20] system combined some ideas from programming by example, constrained-based systems and graphical programming frameworks. It used data abstracts to let user draw data structures graphically and use constraints to specify data structure invariants. The programmer could then manipulate these graphical abstractions and

perform an execution of the program on some example input. This system provided a platform for programmer to program pictorialy but still the problem of reasoning about the precise execution of the program remained.

The Storyboard programming framework combines ideas from visual programming, programming by example and software synthesis research. It lets programmer specify graphical specifications for input-output pairs (potentially infinite due to abstraction). The Storyboard framework is different from previous works in programming by example as it requires no concrete executions on the example inputs from the user. Moreover, our framework requires a control flow sketch of the program to be provided by the programmer to structure the program search space and not let the synthesizer synthesize arbitrary programs. This idea of providing programmer's insights helps rule out a large subset of undesirable programs. The framework is moreover different from the previous work in visual programming as it does not require programmers to provide a pictorial execution of the desired program. Only the input and output pictures are required which are much easier to reason about rather than the complete program execution.

Chapter 9

Conclusions and Future Work

In this thesis we present a framework for automatically synthesizing correct implementations of data structure manipulations with unbounded guarantees from graphical specifications. Graphical specifications with combined synthesis engine alleviate the task of complex reasoning about the corresponding low level implementation for the programmers. Programmer only needs to reason graphically about the manipulation which is much more intuitive than the low level code. The thesis also presents a novel algorithm to automatically derive abstract domain from the storyboards that is required to provide unbounded correctness guarantees for the synthesized implementation. We present preliminary results for the feasibility of our approach on the linked list and binary search tree manipulation examples.

Our immediate future work is to implement the graphical frontend for the storyboard language. We intend to provide basic graphical constructs similar to Thinkpad [20] for programmers to draw abstract data structures graphically. We also intend to allow programmers to specify constraints over the data abstractions in the frontend. We then plan to use our framework for synthesizing challenge benchmarks like red black tree manipulations. And ultimately, our goal is for programmers to use this framework for multitude of tasks involving manipulations for arbitrary user defined data structures.

We also plan to investigate the approach of modular synthesis using storyboards. For synthesizing large implementations, we intend to first synthesize small modules using storyboards and then search over these modules to get the desired large implementations. For example insertion in a red black tree can be broken down to two modules: a module for insertion in binary search tree and a module fixing the height invariant. Furthermore, the binary search tree module can be used as a sub-module for fixing the height invariant module.

Bibliography

[1] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI*, pages 203–213, 2001.

[2] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *PLDI*, pages 300–309, 2007.

[3] Alan Borning. The programming language aspects of thinglab, a constraint-oriented simulation laboratory. *ACM Trans. Program. Lang. Syst.*, 3(4):353–387, 1981.

[4] Marc H. Brown and Robert Sedgewick. Techniques for algorithm animation. *IEEE Software*, 2(1):28–39, 1985.

[5] Carlos Christensen. On the implementation of ambit, a language for symbol manipulation. *Commun. ACM*, 9(8):570–573, 1966.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2001.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[8] T. O. Ellis, J. F. Heafner, and W. F. Sibley. The grail project: An experiment in man-machine communications. Technical Report RM-5999-ARPA, RAND, 1969.

[9] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 281–292, New York, NY, USA, 2008. ACM.

[10] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Constraint-based invariant inference over predicate abstraction. In *VMCAI '09: Proceedings of the 2009 Conference on Verification Model Checking and Abstract Interpretation*, pages 120–135, 2009.

[11] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *CAV*, pages 226–238, 2005.

[12] Gary A. Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, New York, NY, USA, 1973. ACM.

[13] Henry Lieberman. Tinker: Example-based programming for artificial intelligence. In *IJCAI*, page 1060, 1981.

[14] Z. Manna and R. Waldinger. Synthesis: Dreams => programs. *IEEE Transactions on Software Engineering*, 5(4):294–328, 1979.

[15] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.

[16] B. A. Myers. Displaying data structures for interactive debugging. Technical Report CSL-80-7, Xerox PARC, 1980.

[17] B. A. Myers. Visual programming, programming by example, and program visualization: a taxonomy. *SIGCHI Bull.*, 17(4):59–66, 1986.

[18] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[19] Amir Pnueli and Roni Rosner. On the synthesis of an asynchronous reactive module. In *ICALP '89: Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, pages 652–671, London, UK, 1989. Springer-Verlag.

[20] R.V. Rubin, E.J. Colin, and S.P. Reiss. Think pad: A graphical system for program-ming by demonstration. *IEEE Software*, 2:73–79, 1985.

[21] David E. Shaw, William R. Swartout, and C. Cordell Green. Inferring lisp programs from examples. In *IJCAI'75: Proceedings of the 4th international joint conference on Artificial intelligence*, pages 260–267, San Francisco, CA, USA, 1975. Morgan Kaufmann Publishers Inc.

[22] David Canfield Smith. *Pygmalion: a creative programming environment.* PhD thesis, Stanford, CA, USA, 1975.

[23] Armando Solar-Lezama. *Program Synthesis By Sketching.* PhD thesis, EECS Dept., UC Berkeley, 2008.

[24] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, volume 42, pages 167–178, New York, NY, USA, 2007. ACM.

[25] Armando Solar-Lezama, Chris Jones, Gilad Arnold, and Rastislav Bodík. Sketching concurrent datastructures. In *PLDI 08*, 2008.

[26] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodik, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 281–294, New York, NY, USA, 2005. ACM Press.

[27] Saurabh Srivastava, Sumit Gulwani, and Jeffrey Foster. From program verification to program synthesis. *POPL*, 2010.

[28] Ivan E. Sutherland. *Sketchpad, A Man-Machine Graphical Communication System.* Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York, 1963.

[29] Martin Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. *SIGPLAN Not.*, 43(6):125–135, 2008.

[30] Martin Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, New York, NY, USA, 2010. ACM.

[31] Jan Wielemaker. An overview of the swi-prolog programming environment. In *WLPE*, pages 1–16, 2003.